GNU TeXmacs user manual

Table of contents

Chapter 1

Getting started

1.1. Conventions for this manual

Throughout the TeXmacs manual, menu entries will be typeset using a *sans serif* font, like in Document, File → Load or Text → Font shape → Italic. Keyboard input will be typeset in a *typewriter* font inside boxes, like in `C-s`. At the righthand side of menu entries, you see keystroke equivalents, when these are available. The following abbreviations are used for such keystrokes:

`S-`. For shift key combinations.

`C-`. For control key combinations.

`A-`. For alternate key combinations.

`M-`. For meta key combinations.

`H-`. For hyper key combinations.

For instance, `A-C-b` stands for `alternate-control-b`. Spaces inside keyboard shortcuts indicate multiple key-presses. For instance, `M-t N b` stands for `meta-t` `N` `b`.

The `alternate`, `meta` and `hyper` keys are not available on all keyboards. On recent PC's, the `meta` key is often replaced by the `windows` key. In the case when one or several modifier keys are missing on your keyboard, you may use `escape` instead of `M-`, `escape escape` instead of `A-` and `F5`, `escape escape escape` or `A-C-` instead of `H-`. For instance, `escape w` is equivalent to `A-w`. You may also configure the keyboard modifiers in order to take full advantage out of the powerful set of keyboard shortcuts which is provided by TeXmacs.

Notice that the TeXmacs menus and keyboard behavior are *contextual*, i.e. they depend on the current mode (i.e. text mode or "math mode"), the current language and the position of the cursor inside your document. For instance, inside math mode, you have special keyboard shortcuts which are handy for typing mathematical formulas, but which are useless in text mode.

1.2. Configuring TeXmacs

When starting TeXmacs for the first time, the program automatically configures itself in a way which it thinks to be most suitable for you. For instance, TeXmacs will attempt to determine your systems settings for the language and the paper type of your printer. However, the automatic configuration may sometimes fail or you may want to use an alternative configuration. In that case, you should go to the Edit → Preferences menu and specify your preferences.

In particular, we recommend you to configure the desired "look and feel" of TeXmacs. By default, we use the Emacs look and feel, which ensures a limited compatibility of the TeXmacs keyboard shortcuts with those of Emacs. Also, TeXmacs comes with a powerful keyboard shortcut system, which attempts to optimize the use of the modifier keys like `shift` and `control` on your keyboard. However, on many X Window systems these modifier keys are not well configured, so that you may wish to redo this yourself. More details can be found in the section about the configuration of TeXmacs.

1.3. Creating, saving and loading documents

When launching TeXmacs without any command line options, the editor automatically creates a new document for you. You may also create a new document yourself using File → New. Newly created documents do not yet carry a name. In order to give them a name, you should click on File → Save as.

We recommend you to give documents a name immediately after their creation; this will avoid you to loose documents. It is also recommended to specify the global settings for your document when necessary. First of all, you may specify a document style like article, book or seminar using Document → Style. If you write documents in several languages, then you may want to specify the language of your document using Document → Language. Similarly, you may specify a paper type using Document → Page → Size.

After modifying your document, you may save it using File → Save. Old documents can be retrieved using File → Load. Notice that you can edit several documents in the same window using TeXmacs; you can switch between different *buffers* using Go.

1.4. Printing documents

You can print the current file using File → Print → Print all. By default, TeXmacs assumes that you have a 600dpi printer for a4 paper. These default settings can be changed in Edit → Preferences → Printer. You can also print to a postscript file using File → Print → Print all to file (in which case the default printer settings are used for creating the output) or File → Export → Postscript (in which case the printer settings are ignored).

You may export to PDF using File → Export → Pdf. Notice that you should set Edit → Preferences → Printer → Font type → True Type if you want the produced Postscript or PDF file to use True Type fonts. However, only the CM fonts admit True Type versions. These CM fonts are of a slightly inferior quality to the EC fonts, mainly for accented characters. Consequently, you might prefer to use the EC fonts as long as you do not need a PDF file which looks nice in Acrobat Reader.

When adequately configuring TeXmacs, the editor is guaranteed to be *wysiwyg*: the result after printing out is exactly what you see on your screen. In order to obtain full wysiwygness, you should in particular select Document → Page → Type → Paper and Document → Page → Screen layout → Margins as on paper. You should also make sure that the characters on your screen use the same number of dots per inch as your printer. This rendering precision of the characters may be changed using Document → Font → Dpi. Currently, minor typesetting changes may occur when changing the dpi, which may globally affect the document through line and page breaking. In a future release this drawback should be removed.

Chapter 2
Writing simple documents

2.1. Generalities for typing text

As soon as you have performed the preparatory actions as explained above, you can start typing. The usual English characters and punctuation symbols can easily be obtained on most keyboards. Accented characters from foreign languages can systematically be obtained using the escape key. For instance, "é" is obtained by typing `A-' e`. Similarly, we obtain "à" via `` A-` a `` and so on. Long words at borders of successive lines are automatically hyphenated. In order to hyphenate foreign languages correctly, you should specify the language of the document in the menu Document → Language.

At the left hand side of the footer, you see the document style, the text properties at the current cursor position. Initially, it displays "generic text roman 10", which means that you type in text mode using a 10 point roman font and the generic document style. You can change the text properties (font, font size, color, language) in the Format menu. You can also change the text properties of the text you have already typed by selecting a region and then using the Format menu. Some text properties can also be changed for all the document with the Document menu.

At the right hand side of the footer, the character or object (like a change in the text properties) just before the cursor is displayed. We also display all environments which are active at the cursor position. This information should help you to orient yourself in the document.

2.2. Typing structured text

Usually, long documents have a structure: they are organized in chapters, sections and subsections, they contain different types of text, such as regular text, citations, footnotes, theorems, etc. After selecting a *document style* in Document → Style, TeXmacs takes care of specific layout issues, such as numbering of sections, pages, theorems, typesetting citations and footnotes in a nice way and so on.

Currently, four standard document styles have been implemented: letter, article, book and seminar. The seminar style is used for making transparencies. As soon as you have selected such a style, you can organize your text into sections (see Text → Section) and use specific *environments*. Examples of environments are theorem, proposition, remark and so on (see Text → Environment). Other examples are lists of items (see Text → Itemize) or numbered lists (see Text → Enumerate).

When you get more acquainted with TeXmacs, it is possible to add your own new environments in your own style file. Assume for instance that you often make citations and that you want those to appear in italic, with left and right margins of 1cm. Instead of manually changing the text and paragraph properties each time you make a citation, it is better to create a citation environment. Not only it will be faster to create a new citation when doing so, but it is also possible to systematically change the layout of your citations throughout the document just by changing the definition of the citation environment. The latter situation occurs for instance if you discover *a posteriori* that you prefer the citations to appear in a smaller font.

2.3. Content-based tags

The simplest examples of structure in a text are content-based tags. In Text → content tags you see a list of them. Content based tags indicate that a given portion of text is of a particular kind or that it serves a specific purpose. For instance, important text should be marked using the strong tag. Its default rendering uses a bold type face, like in this **strong text**. However, strong text might be rendered in a different way according to the document style. For instance, strong text may be rendered in a different color on transparencies for presentations. Here follows a short list of the most common content-based tags and their purpose:

Tag	Example	Purpose
strong	this is **important**	Indicate an important region of text
em	the *real* thing	Emphasize a region of text
dfn	A *gnu* is a horny beast	Definition of some concept
samp	the `ae` ligature æ	A sequence of literal characters
name	the LINUX system	The name of a particular thing
person	I am JORIS	The name of a person
cite*	Melville's *Moby Dick*	A bibliographic citation
abbr	I work at the C.N.R.S.	An abbreviation
acronym	the HTML format	An acronym
verbatim	the program said `hello`	Verbatim text like computer program output
kbd	Please type `return`	Text which should be entered on a keyboard
code*	`cout << 1+1;` yields 2	Code of a computer program
var	`cp` *`src-file dest-file`*	Variables in a computer program

Table 2.1. Some of the most common content-based tags.

2.4. Lists

Using Text → Itemize you may start an unnumbered list. You may either select a particular tag like • (bullets), − (dashes) or → (arrows) to indicate entries in the list or the default tag. Lists may be *nested* inside other tags, like in the following list:

- First item.
- Now comes the sublist:
  - A subitem.
  - Another one.
- A final item.

The default tag is rendered in a different way depending on the level of nesting. At the outermost level, we used the • tag, at the second level ◦, and so on. When you are inside a list, notice that pressing `return` automatically starts a new item. If you need items which are several paragraphs long, then you may always use `S-return` in order to start a new paragraph.

Enumerate environments, which are started using Text → Enumerate, behave in a similar way as itemize, except that the items are numbered. Here follows an example of an enumeration which was started using Text → Enumerate → Roman:

I. A first item.

II. A second one.

III. And a last one.

The last type of lists are descriptive lists. They are started using Text → Description and allow you to describe a list of concepts:

Gnu. A hairy but gentle beast.

Gnat. Only lives in a zoo.

2.5. Environments

In a similar way as content-based tags, environments are used to mark portions of text with a special meaning. However, while content-based tags usually enclose small portions of text, environments often enclose portions that are several paragraphs long. Frequently used environments in mathematics are theorem and proof, like in the example below:

Theorem 2.1. *There exist no positive integers a, b, c and n with $n \geqslant 3$, such that $a^n + b^n = c^n$.*

Proof. I do not have room here to write the proof down. □

You may enter environments using Text → Environment. Other environments with a similar rendering as theorems are proposition, lemma, corollary, axiom, definition. You may use the dueto macro (entered using `\ d u e t o return`) in order to specify the person(s) to which the theorem is due, like in

Theorem 2.2. (Pythagoras) *Under nice circumstances, we have $a^2 + b^2 = c^2$.*

Other frequently used environments with a similar rendering as theorems, but which do not emphasize the enclosed text, are remark, note, example, warning, exercise and problem. The remaining environments verbatim, code, quote, quotation and verse can be used in order to enter multiparagraph text or code, quotations or poetry.

2.6. Layout issues

As a general rule, TeXmacs takes care of the layout of your text. Therefore, although we did not want to forbid this possibility, we do not encourage you to typeset your document visually. For instance, you should not insert spaces or blank lines as substitutes for horizontal and vertical spaces between words and lines; instead, additional space should be inserted explicitly using Insert → Space. This will make your text more robust in the sense that you will not have to reconsider the layout when performing some minor changes, which affect line or page breaking, or major changes, such as changing the document style.

Several types of explicit spacing commands have been implemented. First of all, you can insert rigid spaces of given widths and heights. Horizontal spaces do not have a height and are either stretchable or not. The length of a stretchable spaces depends on the way a paragraph is hyphenated. Furthermore, it is possible to insert tabular spaces. Vertical spaces may be inserted either at the start or the end of a paragraph: the additional vertical space between two paragraphs is the maximum of the vertical space after the first one and the vertical space before the second one (contrary to TEX, this prevents from superfluous space between two consecutive theorems).

As to the paragraph layout, the user may specify the paragraph style (justified, left ragged, centered or right ragged), the paragraph margins and the left (resp. right) indentation of the first (resp. last) line of a paragraph. The user also controls the spaces between paragraphs and successive lines in paragraphs.

You can specify the page layout in the Document → Page menu. First of all, you can specify the way pages are displayed on the screen: when selecting "paper" as page type in Document → Page → Type, you explicitly see the page breaks. By default, the page type is "papyrus", which avoids page breaking during the preparation of your document. The "automatic" page type assumes that your paper size is exactly the size of your window. The page margins and text width are specified in Document → Page → Layout. Often, it is convenient to reduce the page margins for usage on the screen; this can be done in Document → Page → Screen layout.

2.7. The font selection system

In TEXMACS, fonts have five main characteristics:

- Its name (roman, pandora, concrete, etc.).
- Its family (roman, typewriter or sans serif).
- Its size (a base size (in points) and a relative size (normal, small, etc.).
- Its series (bold, medium or light).
- Its shape (right, italic, small caps, etc.).

Notice that in the font selection system of LATEX 2ε, the font name and family are only one (namely, the family). Notice also that the base font size is specified for the entire document in Document → Font → Size.

2.8. Mastering the keyboard

2.8.1. General prefix rules

Since there are many keyboard shortcuts, it is important to have some ways of classifying them in several categories, in order to make it easier to memorize them. As a general rule, keyboard shortcuts which fall in the same category are identified by a common prefix. The main such common prefixes are:

C-x. Control key based shortcuts are used for frequently used editing commands. They depend very much on the "look and feel" in Edit → Preferences. For instance, if you use an EMACS-compatible look and feel, then the shortcuts of the form C-x correspond to EMACS commands, like C-y for pasting text.

`A-x`. The alternate key is used for commands which depend on the mode that you are in. For instance, `A-s` produces **strong** text in text mode and a square root $\sqrt{}$ in math mode. Notice that `escape escape` is equivalent to `A-`.

`M-x`. The meta key is used for general purpose TeXmacs commands, which can be used in all modes. For instance, `M-!` produces a label. It is also used for additional editing commands, like `A-w` for copying text if you use the Emacs look and feel. Notice that `escape` is equivalent to `M-`.

`H-x`. The user keyboard modifier key is used for producing special symbols like Greek characters in math mode. You may configure your keyboard so as to let caps-lock play the rôle of the hyper key. The `F5` is equivalent to `H-`.

We recall that the particular modifier keys which are used in order to obtain the `M-` and `H-` prefixes can be configured in Edit → Preferences.

2.8.2. Some fundamental keyboard shortcuts

Some standard keyboard actions which are valid in all modes are:

`S-return`. always starts a new paragraph.

`C-backspace`. remove the containing object or environment.

`A-space`. insert a small space.

`A-S-space`. insert a small negative space.

`M-A-home`. manually set start of the selection.

`M-A-end`. manually set end of the selection.

`M-<`. go to the start of the document.

`M->`. go to the end of the document.

2.8.3. Keyboard shortcuts for text mode

To write a text in an european language with a keyboard which does have the appropriate special keys, you can use the following shortcuts to create accented characters. Note that they are active regardless of the current language setting.

Shortcut		Example		Shortcut		Example	
`A-'`	Acute ´	`A-' e`	é	`` A-` ``	Grave `	`` A-` e ``	è
`A-^`	Hat ˆ	`A-^ e`	ê	`A-"`	Umlaut ¨	`A-" e`	ë
`A-~`	Tilde ˜	`A-~ a`	ã	`A-C`	Cedilla ¸	`A-C c`	ç
`A-U`	Breve ˘	`A-U g`	ğ	`A-V`	Check ˇ	`A-V s`	š
`A-O`	Above ring ˚	`A-O a`	å	`A-.`	Above dot ˙	`A-. z`	ż
`A-H`	Hungarian ˝	`A-H o`	ő				

Table 2.2. Typing accented characters.

Special characters can also be created in any language context:

Shortcuts							
S-F5 a	æ	S-F5 A	Æ	S-F5 a e	æ	S-F5 A E	Æ
S-F5 o	ø	S-F5 O	Ø	S-F5 o e	œ	S-F5 O E	Œ
S-F5 s	ß	S-F5 S	SS				
S-F5 !	¡	S-F5 ?	¿	S-F5 p	§	S-F5 P	£

Table 2.3. Typing special characters.

When you press the " key, an appropriate quote will be inserted. The quote character is chosen according to the current language and the surrounding text. If the chosen quoting style is not appropriate, you can change it in Edit → Preferences → Keyboard → Automatic quotes. You can also insert raw quotes:

Shortcuts			
S-F5 "	"	, ,	„
< tab	‹	> tab	›
< <	«	> >	»

Table 2.4. Typing raw quotes.

"English" quotes are considered ligatures of two successive backticks or apostrophes. They can be created with ‘ ‘ and ’ ’ but these are not actual keyboard commands: the result is two characters displayed specially, not a special single character.

Some shortcuts are available in specific language contexts. You can set the text language for the whole document with Document → Language or only locally with Format → Language (see generalities for typing text).

Hungarian		Spanish		Polish			
S-F5 o	ő	! tab	¡	S-F5 a	ą	S-F5 o	ó
S-F5 O	Ő	? tab	¿	S-F5 A	Ą	S-F5 O	Ó
S-F5 u	ű	! ‘	¡	S-F5 c	ć	S-F5 s	ś
S-F5 U	Ű	? ‘	¿	S-F5 C	Ć	S-F5 S	Ś
				S-F5 e	ę	S-F5 x	ź
				S-F5 E	Ę	S-F5 X	Ź
				S-F5 l	ł	S-F5 z	ż
				S-F5 L	Ł	S-F5 Z	Ż
				S-F5 n	ń	S-F5 z tab	ź
				S-F5 N	Ń	S-F5 Z tab	Ź

Table 2.5. Language-specific text shorthands.

Language-specific shortcuts override generic shortcuts; for example, you cannot easily type "ø" in hungarian context.

2.8.4. Hybrid commands and LaTeX simulation

TeXmacs allows you to enter LaTeX commands directly from the keyboard as follows. You first hit the `\`-key in order to enter the hybrid LaTeX/TeXmacs command mode. Next you type the command you wish to execute. As soon as you finished typing your command, the left footer displays something like

```
<return>: action to be undertaken
```

When you hit the `return`-key at this stage, your command will be executed. For instance, in math-mode, you may create a fraction by typing `\ f r a c return`.

If the command you have typed is not a (recognized) LaTeX command, then we first look whether the command is an existing TeXmacs macro, function or environment (provided by the style file). If so, the corresponding macro expansion, function application or environment application is created (with the right number of arguments). Otherwise, it is assumed that your command corresponds to an environment variable and we ask for its value. The `\`-key is always equivalent to one of the commands `M-i l`, `M-i e`, `M-i a`, `M-i #` or `M-i v`.

To insert a literal \ (backslash) character, you can use the `S-F5 \` sequence.

2.8.5. Dynamic objects

Certain more complex objects can have several *states* during the editing process. Examples of such *dynamic objects* are labels and references, because the appearance of the reference depends on a dynamically determined number. Many other examples of dynamic markup can be found in the documentation about writing style files.

When entering a dynamic object like a label using `M-!`, the default state is *inactive*. This inactive state enables you to type the information which is relevant to the dynamic object, such as the name of the label in our case. Certain dynamic objects take an arbitrary number of parameters, and new ones can be inserted using `tab`.

⟨label|pythagoras⟩

Figure 2.1. Inactive label

When you finished typing the relevant information for your dynamic object, you may type `return` in order to *activate* the object. An active dynamic object may be deactivated by placing your cursor just behind the object and hitting `backspace`.

2.8.6. Customization of the keyboard

It is possible for the user to modify the keyboard behaviour. In order to do so, we suggest first to look at the files in the directory `$TEXMACS_PATH/progs/keyboard`, where the standard keyboard behaviour is defined. Then you may redefine the keyboard behaviour in your private initialization file.

Chapter 3

Mathematical formulas

To type mathematical formulas, you need first to enter “math mode”. This is a special text property enabled in structures created by the Text → Mathematics menu.

Formula `$`**.** is used for small mathematical fragments inside a textual paragraph.

Note that formulas are typeset specially so they do not take too much vertical space. For example, limits are always displayed on the left. Limits can be displayed below in formulas with Format → Formula style → on. In formulas, formula style is off by default.

Equation `A-$`**.** is the structure for bigger mathematical expressions which are typeset in a paragraph of their own.

Equations `A-&`**.** create an eqnarray*, a three columns wide table-like environment (see creating tables).

This environment should be used for multiple relations where each line repeats the relation symbol. The first column should contain the left hand side, the middle column the relational symbol, and the left column the right hand side. The typical use for eqnarray* is a step by step computation where each line describes a simple operation on the right hand side of an equation.

In math mode, you have specific commands and key-combinations to type mathematical symbols and formulas. For instance, the `H-` prefix can be used in order to enter Greek symbols (recall that `H-` is equivalent to `F5`, `escape escape escape` or `A-C-`).

The editor favors typing mathematics with a certain meaning. This feature, which will be developed more in future releases, is useful when communicating with a computer algebra package. At this moment, you should for instance explicitly type the multiplication `*` between symbols a and b. By default, typing `a b` will yield ab and not $a\,b$.

3.1. Main mathematical constructs

The main mathematical objects are created using the `A-` prefix as follows:

Shortcut	Purpose	Example
`A-$`	Text	$L=\{x\|x \text{ is sufficiently large}\}$
`A-f`	Fractions	$\frac{a}{b+c}$
`A-s`	Square roots	$\sqrt{x+y}$
`A-S`	n-th Roots	$\sqrt[3]{x^3+y^3}$
`A-n`	Negations	$\frac{a\!\!\!/}{b\!\!\!/\!+c}$

Table 3.1. Creation of major mathematical markup.

Primes, subscripts and superscripts are created as follows:

Shortcut	Purpose	Example
`'`	Primes	f' or $(g+h)'''$
`` ` ``	Back-primes	${}^{\backprime}f$
`_`	Subscripts	x_n or x_{i_3}
`^`	Superscripts	x^2, x_n^2 or $\mathrm{e}^{\mathrm{e}^x}$
`A-l _`	Left subscripts	${}_2x$
`A-l ^`	Left superscripts	${}^{\pi}x$ or ${}^{*}_{*}\mathrm{He}^{*}_{*}$

Table 3.2. Creation of primes, subscripts and superscripts

Some important mathematical constructs are actually tabular constructs and are documented separately.

3.2. Typing mathematical symbols

The Greek characters are obtained in $\mathrm{T_EX_{MACS}}$ by combining the hyper modifier key `H-` with a letter. For instance, `H-a` yields α and `H-G` yields Γ. Recall that the `F5` key is equivalent to `H-`, so that ρ can also be obtained by typing `F5 r`. Similarly, `F6`, `F7`, `F8` and `S-F6` can be used in order to type bold, calligraphic, fraktur and blackboard bold characters. For instance, `F8 m` yields $\mathfrak{m}$, `S-F6 R` yields $\mathbb{R}$ and `F6 F7 Z` yields $\boldsymbol{\mathcal{Z}}$.

Greek characters can also be obtained as "variants" of Latin characters using the `tab`-key. For instance, `p tab` yields π. The `tab`-key is also used for obtaining variants of the Greek letters themselves. For instance, both `H-p tab` and `p tab tab` yield ϖ.

Many other mathematical symbols are obtained by "natural" key-combinations. For instance, `- >` yields $\rightarrow$, `- - >` yields $\longrightarrow$ and `> =` yields $\geqslant$. Similarly, `| -` yields $\vdash$, `| - >` yields $\mapsto$ and `- > < -` yields $\rightleftarrows$. Some general rules hold in order to obtain variants of symbols:

`tab`. is the main key for obtaining variants. For instance, `> =` yields $\geqslant$, but `> = tab` yields $\geq$. Similarly, `< tab` yields $\prec$, `< tab =` yields $\preccurlyeq$ and `< tab = tab` yields $\preceq$. Also, `P tab` yields $\wp$ and `e tab` yields the constant $\mathrm{e} = \exp(1)$. You may "cycle back" using `S-tab`.

`@`. is used for putting symbols into circles or boxes. For instance, `@ +` yields $\oplus$ and `@ x` yields $\otimes$. Similarly, `@ tab +` yields $\boxplus$.

`/`. is used for negations. For instance, `= /` yields $\neq$ and `< = /` yields $\nleqslant$. Notice that `< = tab tab /` yields $\nleq$, while `< = tab tab / tab` yields $\lneq$.

`!`. is used after arrows in order to force scripts to be placed above or below the arrow. For instance, `- - > ^ x` yields $\longrightarrow^x$, but `- - > ! ^ x` yields $\xrightarrow{x}$.

Several other symbols which cannot be entered naturally in the above way are obtained using the `S-F5` prefix. Here follows a short table of such symbols:

Shortcut	Symbol	Shortcut	Symbol
`S-F5 a`	$\amalg$		
`S-F5 n`	$\cap$	`S-F5 u`	$\cup$
`S-F5 v`	$\vee$	`S-F5 w`	$\wedge$

Table 3.3. Some symbols which cannot be obtained using general rules in a natural way.

3.3. Typing big operators

The following key-combinations are used in order to create big symbols:

Shortcut	Result	Shortcut	Result
`S-F5 I`	$\displaystyle\int$	`S-F5 O`	$\displaystyle\oint$
`S-F5 P`	$\displaystyle\prod$	`S-F5 A`	$\displaystyle\coprod$
`S-F5 S`	$\displaystyle\sum$	`S-F5 @ +`	$\displaystyle\bigoplus$
`S-F5 @ x`	$\displaystyle\bigotimes$	`S-F5 @ .`	$\displaystyle\bigodot$
`S-F5 U`	$\displaystyle\bigcup$	`S-F5 N`	$\displaystyle\bigcap$
`S-F5 V`	$\displaystyle\bigvee$	`S-F5 W`	$\displaystyle\bigwedge$

Table 3.4. Big mathematical operators.

The big integral signs admit two variants, depending on where you want to place subscripts and superscripts. By default, the scripts are placed as follows:

$$\int_0^\infty \frac{\mathrm{d}x}{1+x^2}.$$

The alternative rendering "with limits"

$$\int\limits_0^\infty \frac{\mathrm{d}x}{1+x^2}.$$

is obtained using `S-F5 L I`. Similarly, you may type `S-F5 L O` in order to obtain $\oint$ with limits.

3.4. Typing large delimiters

Large delimiters are created as follows:

Shortcut	Result	Shortcut	Result
A-(	$($	A-)	$)$
A-[	$[$	A-]	$]$
A-{	$\{$	A-}	$\}$
A-<	$\langle$	A->	$\rangle$
A-/	$/$	A-\	$\backslash$

Table 3.5. Keyboard shortcuts for large delimiters.

In TeXmacs, large delimiters may either be "left delimiters", "right delimiters" or "middle delimiters". By default, $($, $[$, $\{$ and $\langle$ are left delimiters, $)$, $]$, $\}$ and $\rangle$ are right delimiters and $|$, $/$ and $\backslash$ are middle delimiters. But there status can be changed using the A-l, A-r and A-m key combinations. For instance, A-l) produces $)$, considered as a large left delimiter.

In TeX and LaTeX, "middle delimiters", or "separators" do not exist; they are used for producing the vertical bars in formulas like

$$\left\langle \frac{a}{b+c} \middle| \frac{p}{q+r} \middle| \frac{a}{b+c} \right\rangle.$$

There may be as many middle delimiters between a left and a right delimiter as one wishes.

3.5. Wide mathematical accents

The table below how to type mathematical accents above symbols or entire formulas. Indeed, some of these accents automatically become as wide as the formulas below them.

Shortcut	Example	Wide variant	Shortcut	Result
A-~	$\tilde{x}$	$\widetilde{x+y}$	A-'	$\acute{x}$
A-^	$\hat{x}$	$\widehat{x+y}$	A-`	$\grave{x}$
A-B	$\bar{x}$	$\overline{x+y}$	A-.	$\dot{x}$
A-V	$\vec{x}$	$\overrightarrow{\mathrm{AB}}$	A-"	$\ddot{x}$
A-C	$\check{x}$	$\widecheck{x+y}$		
A-U	$\breve{x}$	$\widebreve{x+y}$		

Table 3.6. Keyboard shortcuts for wide mathemarical accents.

Chapter 4

Tabular material

4.1. Creating tables

In order to create a table, you may either use Insert → Table or one of the following keyboard shorthands:

`M-t N t`. Create a regular table.

`M-t N T`. Create a regular table whose cells are centered.

`M-t N b`. Create a regular “block”, whose cells are separated by lines.

`M-t N B`. Create a block whose cells are centered.

In math mode, a few other table-like structures are provided:

`M-t N m`. Create a matrix.

`M-t N d`. Create a determinant.

`M-t N c`. Create a choice list.

The eqnarray* environment is also a special kind of table-like structure, which extends over the entire line. You may start a list of equations using Text → Mathematics → Equations.

When starting a new table, its size is minimal (usually 1×1) and its cells are empty. New rows and columns are inserted using the `A-left`, `A-right`, `A-up` and `A-down` shorthands. For instance, `A-right` creates a new column at the right of the current cursor position. You may also start a new row below the current cursor position by hitting `return`.

4.2. The formatting mode

In TeXmacs, arbitrary blocks of cells in the table may be formatted in particular ways. For instance, you may give individual cells a background color, but you may also decide an entire column to be horizontally centered. By default, formatting commands operate on individual cells, but this may be changed via Table → Cell operation mode. The following operation modes are available:

`M-t m c`. Operate on individual cells.

`M-t m h`. Operate on rows.

`M-t m v`. Operate on columns.

`M-t m t`. Operate on the entire table.

It is also possible to select a block of cells using the mouse and perform a single operation on that rectangle.

4.3. Specifying the cell and table alignment

The most frequent formatting operation is the horizontal or vertical alignment of a block of cells. You may use the `M-←`, `M-→`, `M-↑` and `M-↓` keystrokes to quickly align more to the left, right, top or bottom.

A specific alignment can also be selected in the Table → Horizontal cell alignment and Table → Vertical cell alignment menus. Alternatively, you may use keyboard shorthands of the types `M-t h x` and `M-t v x` for horizontal resp. vertical alignment.

Similarly, you may specify how the table itself should be aligned with respect to the surrounding text. This is either done via the Table → Horizontal table alignment and Table → Vertical table alignment submenus, or using keyboard shorthands of the form `M-t H x` or `M-t V x`. Here `x` represents `l` for "left", `c` for "centered", `r` for "right", `b` for "bottom" and `t` for "top".

4.4. Specifying the cell and table size

Using Table → Cell width → Set width resp. Table → Cell height → Set height you may specify the width or height of a cell. In fact, the specified width (or height) may be taken into account in three different ways:

Minimum mode. The actual width of the cell will be the minimum of the specified width and the width of the box inside the cell.

Exact mode. The width of the cell will be precisely the specified one.

Maximum mode. The actual width of the cell will be the maximum of the specified width and the width of the box inside the cell.

The border width and the cell padding (to be explained below) are taken into account in the size of the box inside the cell.

You may also specify the width and the height of the entire table in Table → Special table properties. In particular, you may specify the table to run over the entire width of a paragraph. When specifying a width (or height) for the entire table, you may specify how the unused space is distributed over the cells using Table → Special cell properties → Distribute unused space. By default, the unused space is equally distributed.

4.5. Borders, padding and background color

You may specify the border widths and padding spaces of a cell in all possible four directions: on the left, on the right, at the bottom and at the top (see Table → Cell border). You have keyboard shorthands of the forms `M-t b x` and `M-t p x` in order to specify border widths and cell padding.

The default border width for cells in the block environment is `1ln`, i.e. the standard line width in the current font (like the width of a fraction bar). This width occurs at the right and the bottom of each cell (except when the cell is on the first row or column). The default horizontal cell padding is `1spc`: the width of a white space in the current font. The default vertical cell padding is `1sep`: the standard minimal separation between two close boxes.

Cells may be given a background color via Table → Cell background color.

The entire table may also be given a border and a table padding in Table → Special table properties → Border. In this case, the padding occurs outside the border.

4.6. Advanced table features

In the menus, you also find some other more special features for tables. Very briefly, these include the following:

- Change the "span" of a cell and let it run over its neighbouring cells on its right and below.
- Creation of entire subtables inside cells.
- Correction of the depth and height of text, in order to let the baselines match.
- Horizontal hyphenation of cell contents and vertical hyphenation of the entire table.
- Gluing several rows and/or columns together, so that the glued cells become "part of the borders" of the remaining cells.
- Disactivation of the table, in order to see its "source code".
- Setting the "extension center" of a table. From now on, the formatting properties of this cell will be used for new cells created around this center.
- Specification of the minimal and maximum size of a table, which will be respected during further editing. (this is mainly useful when creating table macros).

Currently, all tables come inside an environment like tabular, block, matrix, etc. When creating your own table macros, you may use Table → Special table properties → Extract format to extract the format from a given table.

Chapter 5

Links and automatically generated content

5.1. Creating labels, links and references

You may create a new inactive label using `M-!` or Insert → Link → Label and a reference to this label using `M-?` or Insert → Link → Reference. Be careful to put the label at a point where its number will be correct. When labeling sections, the recommended place is just after the section name. When labeling single equations (created using Insert → Mathematics → Equation), the recommended place is at the start inside the equation. When labeling multiple equations (created using Insert → Mathematics → Equations), you must put the labels just behind the equation numbers. Recall that you may use `A-*` in order to transform an unnumbered environment or equation into a numbered one, and vice versa.

It is possible to create hyperlinks to other documents using `M-i >` or Insert → Link → Hyperlink. The first field of the hyperlink is the associated text, which is displayed in blue when activated. The second field contains the name of a document, which may be on the web. As is usual for hyperlinks, a link of the form `#label` points to a label in the same document and a link of the form `url#label` points to a label in the document located at `url`.

In a similar fashion, an action may be associated to a piece of text or graphics using `M-i *` or Insert → Link → Action. The second field now contains a Guile/Scheme script command, which is executed whenever you double click on the text, after its activation. For security reasons, such scripts are not always accepted. By default, you are prompted for acceptation; this default behaviour may be changed in Options → Security. Notice that the Guile/Scheme command

```
(system "shell-command")
```

evaluates `shell-command` as a shell command.

Finally, you may directly include other documents inside a given document using `M-i i` or Insert → Link → Include. This allows you for instance to include the listing of a program in your text in such a way that your modifications in your program are automatically reflected in your text.

5.2. Inserting images

You can include images in the text using the menu Insert → Image. Currently, $\mathrm{T_EX_{MACS}}$ recognizes the `ps`, `eps`, `tif`, `pdf`, `pdm`, `gif`, `ppm`, `xpm` and `fig` file formats. Here, `gs` (i.e. ghostscript) is used to render postscript images. If ghostscript has not yet been installed on your system, you can download this package from

```
www.cs.wisc.edu/~ghost/index.html
```

Currently, the other file formats are converted into postscript files using the scripts `tiff2ps`, `pdf2ps`, `pnmtops`, `giftopnm`, `ppmtogif`, `xpmtoppm`. If these scripts are not available on your system, please contact your system administrator.

By default, images are displayed at their design size. The following operations are supported on images:

- Clipping the images following a rectangle. The lower left corner of the default image is taken as the origin for specifying a rectangle for clipping.

- Resizing an image. When specifying a new width, but no height at the prompt (or vice versa), the image is resized so as to preserve the aspect ration.

- Magnifying the image. An alternative way to resize an image, by multiplying the width and the height by a constant.

We also included a script to convert pictures, with optional LaTeX formulas in it, into encapsulated postscript. In order to include a LaTeX formula in an `xfig` picture, we recall you should enter the formula as text, while selecting a LaTeX font and setting the special flag in the text flags.

5.3. Generating a table of contents

It is very easy to generate a table of contents for your document. Just put your cursor at the place where you want your table of contents and click on Text → Automatic → Table of contents.

In order to generate the table of contents, you should be in a mode where page breaks are visible (select paper in Document → Page → Type), so that the appropriate references to page numbers can be computed. Next, use Document → Update → Table of contents or Document → Update → All to generate the table of contents. You may have to do this several times, until the document does not change anymore. Indeed, the page numbers may change as a result of modifications in the table of contents!

5.4. Compiling a bibliography

At the moment, TeXmacs uses `bibtex` to compile bibliographies. The mechanism to automatically compile a bibliography is the following:

- Write a `.bib` file with all your bibliographic references. This file should have the format of a standard bibliography file for LaTeX.

- Use Insert → Link → Citation and Insert → Link → Invisible citation to insert citations, which correspond to entries in your `.bib` file.

- At the place where your bibliography should be compiled, click on Text → Automatic → Bibliography. At the prompt, you should enter a `bibtex` style (such as `plain`, `alpha`, `abbrv`, etc.) and your `.bib` file.

- Use Document → Update → Bibliography in order to compile your bibliography.

Notice that additional BiBTeX styles should be put in the directory `~/.TeXmacs/system/bib`.

5.5. Generating an index

For the generation of an index, you first have to put index entries in your document using Insert → Link → Index entry. At a second stage, you must put your cursor at the place where you want your index to be generated and click on Text → Automatic → Index. The index is than generated in a similar way as the table of contents.

In the Insert → Link → Index entry menu, you find several types of index entries. The simplest are "main", "sub", "subsub", which are macros with one, two and three arguments respectively. Entries of the form "sub" and "subsub" may be used to subordinate index entries with respect to other ones.

A complex index entry takes four arguments. The first one is a key how the entry has to be sorted and it must be a "tuple" (created using `M-i <`) whose first component is the main category, the second a subcategory, etc. The second argument of a complex index entry is either blank or "strong", in which case the page number of your entry will appear in a bold typeface. The third argument is usually blank, but if you create two index entries with the same non-blank third argument, then this will create a "range" of page numbers. The fourth argument, which is again a tuple, is the entry itself.

It is also possible to create an index line without a page number using "interject" in Insert → Link → Index entry. The first argument of this macro is a key for how to sort the index line. The second argument contains the actual text. This construct may be useful for creating different sections "A", "B", etc. in your index.

5.6. Compiling a glossary

Glossaries are compiled in a similar way as indexes, but the entries are not sorted. A "regular" glossary entry just contains some text and a page number will be generated for it. An "explained" glossary entry contains a second argument, which explains the notation. A "duplicate" entry may be used to create a page number for the second occurence of an entry. A glossary line creates an entry without a page number.

5.7. Books and multifile documents

When a document gets really large, you may want to subdivide it into smaller pieces. This both makes the individual pieces more easily reusable in other works and it improves the editor's responsiveness. An entire file can be inserted into another one using Insert → Link → Include. In order to speed up the treatment of included documents, they are being buffered. In order to update all included documents, you should use Tools → Update → Inclusions.

When writing a book, one usually puts the individual chapters in files `c1.tm`, `c2.tm` until `cn.tm`. One next creates one file `book.tm` for the whole book, in which the files `c1.tm`, `c2.tm` until `cn.tm` are included using the above mechanism. The table of contents, bibliography, etc. are usually put into `book.tm`.

In order to see cross references to other chapters when editing a particular chapter `ci.tm`, one may specify `book.tm` as a "master file" for the files `c1.tm` to `cn.tm` using Document → Master → Attach. Currently, the chapter numbers themselves are not dealt with by this mechanism, so you may want to manually assign the environment variable *chapter-nr* at the start of each chapter file in order to get the numbering right when editing.

Chapter 6

Editing tools

6.1. Cut and paste

You can select text and formulas by maintaining the left mouse button. In order to delete the selected region, use Edit → Cut. In order to copy the selected region, first click on Edit → Copy. Next, paste it as many times as you want to the location of your cursor, using Edit → Paste. Alternatively, you may copy a selected region using the middle mouse button.

It is also possible to the change text properties of a selected region. For instance, in order to transform some black text in red, you select it using the left mouse button and click on Text → Color → Red. Similarly, if you select a formula and you click on Insert → Mathematics → Fraction, then the formula becomes the numerator of some fraction.

When using the copy and paste mechanism to communicate with other applications, text is copied and pasted using the TeXmacs data format. You may specify other import and export formats using Edit → Import resp. Edit → Export. By default, copying and pasting uses the primary text buffer. Using Edit → Copy to and Edit → Paste from, you may specify as many other buffers as you like.

6.2. Search and replace

You can start searching text by pressing `C-s` or Edit → Search. During a search, the "search string" is displayed at the left hand side of the footer. Each character you type is appended to this search string and the next occurrence of it is surrounded by a red box. When pressing `C-s` a second time during a search, the next occurrence is being searched. A beep indicates that no more occurrences were found in the document; pressing `C-s` will continue the search at the beginning of your document. You may press `backspace` in order to undo key presses during a search.

Usually, text is being searched for in a forward manner, starting from the current cursor position. You may also search backwards, using `C-r`. During a search, only text in the same mode and the same language will be found, as those which are active at the position where you started your search. In other words, when searching an x in math-mode, you will not find any x's in the ordinary text. As a current limitation, the search string can only contain ordinary text and no math-symbols or more complicated structured text.

A query replace is started by pressing `C-=` or Edit → Replace. You are prompted for a string which is to be replaced and the string by which to replace. At each occurrence of the string to be replaced you are prompted and you have to choose between replacing the string (y), not replacing it (n) and replace this and all further occurrences (a). Like in the case of searching, the query-replace command is mode and language sensitive.

6.3. Spell checking

If the program `ispell` has been installed on your system, then you may use it to check your text for misspelled words by pressing `M-$` or Edit → Spell. Notice that you might have to verify that the dictionaries corresponding to the languages in which your texts have been written have been installed on your system; this is usually the case for English.

When you launch the spell checker (either on the whole text or a selected region), you will be prompted at each misspelled word and the footer displays the available options:

a). Accepts the misspelled word and all its future occurrences in the text.

r). Replace the misspelled word by a correction you have to enter.

i). Indicate that the "misspelled" word is actually correct and that it has to be inserted in your personal dictionary.

1-9). Several suggested corrections for your misspelled word.

Notice that `ispell` just checks for misspelled words. No grammatical faults will be detected.

When starting the spell checker, it will use the dictionary of the language which is active at the current cursor position (or the start of a selection). Only text in that language will be checked for. If your document contains text in several languages, then you will have to launch the spell checker once for each language being used.

6.4. Undo and redo

It is possible to gradually undo the changes you made in a document from the moment that you launched TeXmacs. This can be done via Edit → Undo or using the keystrokes `M-[` or `C-/`. Undone changes can be "redone" using Edit → Redo or `M-]`.

In order to save memory, the number of successive actions which can be undone is limited to 100 (by default). It is possible to increase this number by adding a command like

```
(set-maximal-undo-depth 1000)
```

in our personal initialization file (see Help → Scheme). When specifying a negative number as your maximal undo depth, any number of actions can be undone.

Chapter 7
Advanced layout features

7.1. Flows

Complex documents often contain footnotes or floating objects, which appear differently on pages as the main text. In fact, the content of such complex documents use several *flows*, one for the main text, one for the footnotes, one for floats, and still another one for two column text. The different flows are broken across pages in a quite independent way.

In order to insert a footnote, you may use Insert → Page insertion → Footnote. The number of columns of the text may be changed in Paragraph → Number of columns.

7.2. Floating objects

Floating objects are allowed to move on the page independently from the main text. Usually they contain figures or tables which are too large to nicely fit into the main text. A floating object may be inserted using Insert → Page insertion → Floating object.

You may also create a floating object and directly insert a figure or table inside it using Insert → Page insertion → Floating figure resp. Insert → Page insertion → Floating table. However, sometimes you might want to insert several smaller figures or tables inside one floating object. You may do this using Insert → Image → Small figure resp. Insert → Table → Small table.

After creating a floating object, you may control its position using Insert → Position float (when inside the float). You may specify whether you allow the floating object to appear at the top of the page, at the bottom, directly in the text, or on the next page. By default, the float may appear everywhere. However, a floating object will never appear inside the main text at less than three lines from the bottom or the top of a page.

7.3. Page breaking

The page breaking may be controlled very precisely by the user inside Document → Page → Breaking. In the submenu Algorithm, you may specify the algorithm being used. Professional page breaking is best in print, but may slow down the editing when being used interactively in paper mode. Sloppy page breaking is fastest and medium is professional except for multicolumn material, for which the professional algorithm is significantly slower.

You may also allow the page breaking algorithm to enlarge or reduce the length of pages in exceptional cases in the submenu Limits. The stretchability of vertical space between paragraphs and so may be specified in Flexibility. The factor 1 is default; a smaller factor enforces a more rigid spacing, but the quality of the breaks may decrease.

Chapter 8

TeXmacs plug-ins

There are many ways in which TeXmacs can be customized or extended: users may define their own style files, customize the user interface, or write links with extern programs. The plug-in system provides a universal mechanism to combine one or several such extensions in a single package. Plug-ins are both easy to install by other users and easy to write and maintain.

8.1. Installing and using a plug-in

From the user's point of view, a plug-in *myplugin* will usually be distributed on some website as a binary tarball with the name

    myplugin-version-architecture.tar.gz

If you installed TeXmacs yourself in the directory `$TEXMACS_PATH`, then you should unpack this tarball in the directory `$TEXMACS_PATH/plugins`, using

```
tar -zxvf myplugin-version-architecture.tar.gz
```

This will create a *myplugin* subdirectory in `$TEXMACS_PATH/plugins`. As soon as you restart TeXmacs, the plug-in should be automatically recognized. Please read the documentation which comes with your plug-in in order to learn using it.

Remark 8.1. If you did not install TeXmacs yourself, or if you do not have write access to `$TEXMACS_PATH`, then you may also unpack the tarball in `$TEXMACS_HOME_PATH/plugins`. Here we recall that `$TEXMACS_HOME_PATH` defaults to `$HOME/.TeXmacs`. When starting TeXmacs, your plug-in should again be automatically recognized.

Remark 8.2. If the plug-in is distributed as a source tarball like *myplugin-version-*`src.tar.gz`, then you should first compile the source code before relaunching TeXmacs. Depending on the plug-in (read the instructions), this is usually done using

```
cd myplugin; make
```

or

```
cd myplugin; ./configure; make
```

Remark 8.3. In order to upgrade a plug-in, just remove the old version in `$TEXMACS_PATH/plugins` or `$TEXMACS_HOME_PATH/plugins` using

```
rm -rf myplugin
```

and reinstall as explained above.

8.2. Writing your own plug-ins

In order to write a plug-in *myplugin*, you should start by creating a directory

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*

where to put all your files (recall that `$TEXMACS_HOME_PATH` defaults to `$HOME/.TeXmacs`). In addition, you may create the following subdirectories (when needed):

`bin` — For binary files.

`doc` — For documentation (not yet supported).

`langs` — For language related files, such as dictionaries (not yet supported).

`lib` — For libraries.

`packages` — For style packages.

`progs` — For Scheme programs.

`src` — For source files.

`styles` — For style files.

As a general rule, files which are present in these subdirectories will be automatically recognized by TeXmacs at startup. For instance, if you provide a `bin` subdirectory, then

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*`/bin`

will be automatically added to the `PATH` environment variable at startup. Notice that the subdirectory structure of a plug-in is very similar to the subdirectory structure of `$TEXMACS_PATH`.

Example 8.4. The easiest type of plug-in only consists of data files, such as a collection of style files and packages. In order to create such a plug-in, it suffices to create directories

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*`/styles`

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*`/packages`

and to put your style files and packages in the last two directories. After restarting TeXmacs, your style files and packages will automatically appear in the Document → Style and Document → Use package menus.

For more complex plug-ins, such as plug-ins with additional Scheme or C++ code, one usually has to provide a Scheme configuration file

`$TEXMACS_HOME_PATH/plugins/`*`myplugin`*`/progs/init-`*`myplugin`*`.scm`

This configuration file should contain an instruction of the following form

```
(plugin-configure myplugin
  configuration-options)
```

Here the *`configuration-options`* describe the principal actions which have to be undertaken at startup, including sanity checks for the plug-in. In the next sections, we will describe some simple examples of plug-ins and their configuration. Many other examples can be found in the directories

$TEXMACS_PATH/examples/plugins

$TEXMACS_PATH/plugins

Some of these are described in more detail in the chapter about writing new interfaces.

8.3. EXAMPLE OF A PLUG-IN WITH SCHEME CODE

The `world` plug-in.

Consider the `world` plug-in in the directory

$TEXMACS_PATH/examples/plugins

This plug-in shows how to extend TeXmacs with some additional SCHEME code in the file

world/progs/init-world.scm

In order to test the `world` plug-in, you should recursively copy the directory

$TEXMACS_PATH/examples/plugins/world

to `$TEXMACS_PATH/plugins` or `$TEXMACS_HOME_PATH/plugins`. When relaunching TeXmacs, the plug-in should now be automatically recognized (a World menu should appear in the menu bar).

How it works.

The file `init-world.scm` essentially contains the following code:

```
(define (world-initialize)
  (menu-extend texmacs-extra-menu
    (=> "World"
        ("Hello world" (insert-string "Hello world")))))

(plugin-configure world
  (:require #t)
  (:initialize (world-initialize)))
```

The configuration option `:require` specifies a condition which needs to be satisfied for the plug-in to be detected by TeXmacs (later on, this will for instance allow us to check whether certain programs exist on the system). The configuration is aborted if the requirement is not fulfilled.

The option `:initialize` specifies an instruction which will be executed during the initialization (modulo the fulfillment of the requirement). In our example, we just create a new top level menu World and a menu item World → Hello world, which can be used to insert the text “Hello world”. In general, the initialization routine should be very short and rather load a module which takes care of the real initialization. Indeed, keeping the `init-`*`myplugin`*`.scm` files simple will reduce the startup time of TeXmacs.

8.4. EXAMPLE OF A PLUG-IN WITH C++ CODE

The `minimal` plug-in.

Consider the example of the `minimal` plug-in in the directory

$TEXMACS_PATH/examples/plugins

It consists of the following files:

minimal/Makefile

minimal/progs/init-minimal.scm

minimal/src/minimal.cpp

In order to try the plug-in, you first have to recursively copy the directory

$TEXMACS_PATH/examples/plugins/minimal

to $TEXMACS_PATH/progs or $TEXMACS_HOME_PATH/progs. Next, running the Makefile using

make

will compile the program minimal.cpp and create a binary

minimal/bin/minimal.bin

When relaunching TeXmacs, the plug-in should now be automatically recognized.

How it works.

The minimal plug-in demonstrates a minimal interface between TeXmacs and an extern program; the program minimal.cpp is explained in more detail in the chapter about writing interfaces. The initialization file init-minimal.scm essentially contains the following code:

```
(plugin-configure minimal
  (:require (url-exists-in-path? "minimal.bin"))
  (:launch "minimal.bin")
  (:session "Minimal"))
```

The :require option checks whether minimal.bin indeed exists in the path (so this will fail if you forgot to run the Makefile). The :launch option specifies how to launch the extern program. The :session option indicates that it will be possible to create sessions for the minimal plug-in using Text → Session → Minimal.

8.5. Summary of the configuration options for plug-ins

As explained before, the Scheme configuration file *myplugin*/progs/init-*myplugin*.scm of a plug-in with name *plugin* should contain an instruction of the type

```
(plugin-configure myplugin
  configuration-options)
```

Here follows a list of the available *configuration-options*:

(:require *condition*) — This option specifies a sanity *condition* which needs to be satisfied by the plug-in. Usually, it is checked that certain binaries or libraries are present on your system. If the condition fails, then TeXmacs will continue as whether your plug-in did not exist. In that case, further configuration is aborted. The :require option usually occurs first in the list of configuration options.

`(:version` ***`version-cmd`***`)` — This option specifies a Scheme expression ***`version-cmd`*** which evaluates to the version of the plug-in.

`(:setup` ***`cmd`***`)` — This command is only executed when the version of the plug-in changed from one execution of TeXmacs to another one. This occurs mainly when installing new versions of TeXmacs or helper applications.

`(:initialize` ***`cmd`***`)` — This option executes the Scheme expression ***`cmd`***. It usually occurs just after the `:require` option, so that the plug-in will only be configured if the plug-in really exists. For large plug-ins, it is important to keep the file ***`myplugin`***`/progs/init-`***`myplugin`***`.scm` small, because it will be rerun each time you start TeXmacs. In order to reduce the boot time, most Scheme commands of the plug-in therefore occur in separate modules, some of which may be loaded by the initialization command.

`(:launch` ***`shell-cmd`***`)` — This option specifies that the plug-in is able to evaluate expressions over a pipe, using a helper application which is launched using the shell-command ***`shell-cmd`***.

`(:link` ***`lib-name export-struct options`***`)` — This option is similar to `:launch`, except that the extern application is now linked dynamically. For more information, see the section about dynamic linking.

`(:session` ***`menu-name`***`)` — This option indicates that the plug-in supports an evaluator for interactive shell sessions. An item ***`menu-item`*** will be inserted to the Text → Session menu in order to launch such sessions.

`(:serializer ,`***`fun-name`***`)` — If the plug-in can be used as an evaluator, then this option specifies the Scheme function ***`fun-name`*** which is used in order to transform TeXmacs trees to strings.

`(:commander ,`***`fun-name`***`)` — This command is similar to the `:serializer` option except that it is used to transform special commands to strings.

`(:tab-completion #t)` — This command indicates that the plug-in supports tab-completion.

`(:test-input-done #t)` — This command indicates that the plug-in provides a routine for testing whether the input is complete.

Chapter 9
Using GNU $\text{T}_\text{E}\text{X}_{\text{MACS}}$ as an interface

An important feature of $\text{T}_\text{E}\text{X}_{\text{MACS}}$ is it's ability to communicate with extern systems in shell-like sessions. Typically, it is possible to evaluate commands of an extern computer algebra system inside such a session and display the results in a nice, graphical way. It is also possible to evaluate shell commands and Scheme programs inside such sessions.

9.1. Creating sessions

A session can be started from the Text → Session menu. A session consists of a sequence of input and output fields and possible text between them. When pressing `return` inside an input field of a session, the text inside the environment is evaluated and the result is displayed in an output field.

When entering a command in a session, the application attempts to execute it. Several commands may be launched concurrently in the same document, but the output will only be active in the session where the cursor is and at the place of the cursor. Therefore, we recommend to use different buffers for parallel executions.

For each type of extern application, one may choose between sharing a single process by different sessions, or launching a separate process for each different session. More precisely, when inserting a session using Text → Session → Other, you may specify both a "session type" (Shell, Pari, Maxima, etc.) and a "session name" (the default name is "default"). Sessions with different names correspond to different processes and sessions with the same name share a common process.

In order to finish the process which underlies a given session, you may use Session → Close session. When pressing `return` in the input of a non-connected system, the system will be restarted automatically. You may also use Session → Interrupt execution in order to interrupt the execution of a command. However, several applications do not support this feature.

9.2. Editing sessions

Inside input fields of sessions, the cursor keys have a special meaning: when moving upwards or downwards, you will move to previous or subsequent input fields. When moving to the left or to the right, you will never leave the input field; you should rather use the mouse for this.

Some facilities for editing input, output and text fields are available in the Session → Insert fields and Session → Remove fields menus. Most operations directly apply to matching input/output fields. Optionally, an additional explanatory text field can be associated to an input field using Session → Insert fields → Insert text field. Keyboard shortcuts for inserting fields are `A-up` (insert above) and `A-down`. Keyboard shortcuts for removing matching text/input/output fields are `A-backspace` (remove backwards) and `A-delete` (remove current fields).

It is possible to create "subsessions" using Session → Insert fields → Fold input field or `A-right`. In that case, the current text/input/output field becomes the body of an unfolded subsession. Such a subsession consists of an explanatory text together with a sequence of text/input/output fields. Subsessions can be folded and unfolded using `M-A-up` resp. `M-A-down`. Subsessions have a nice rendering on the screen when using the `varsession` package in Document → Use package → Program.

Other useful editing operations for text/input/output fields are Session → Remove fields → Remove all output fields, which is useful for creating a demo sessions which will be executed later on, and Session → Split session, which can be used for splitting a session into parts for inclusion into a paper.

9.3. Selecting the input method

By default, TeXmacs will attempt to evaluate the input field when pressing `return`. Multiline input can be created using `S-return`. Alternatively, when selecting the multiline input mode using Session → Input mode → Multiline input, the `return` key will behave as usual and `S-return` may be used in order to evaluate the input field. Notice finally that certain systems admit built-in heuristics for testing whether the input has been completed; if not, then the `return` may behave as usual.

Certain applications allow you to type the mathematical input in a graphical, two dimensional form. This feature can be used by selecting Session → Input mode → Mathematical input. If this feature is available, then it is usually also possible to copy and paste output back into the input. However, it depends on the particular application how well this works.

9.4. Supported systems

When taking a look at the Insert → Session menu, only those systems which are actually installed on your system will show up. The only exceptions are shell sessions and scheme sessions, which are always available.

Below, you find a short list of free computer algebra systems which have been interfaced with TeXmacs. There also exist interfaces with several proprietary interfaces, but you should look at the documentation of those systems for more information.

9.4.1. Shell sessions and scheme sessions

In a "shell session" it is possible to evaluate shell commands. All input and output is verbatim. No particular command-line utilities (such as completion mechanisms) have been implemented yet. The output of the shell command is displayed gradually as the program executes.

In a "Scheme session" you can evaluate Guile/Scheme programs. The input should be verbatim text. The input is evaluated and the result is displayed. No gradual output mechanism has been implemented yet for Scheme session.

9.4.2. Giac

Giac Is A Computer algebra system, which can be downloaded from http://www-fourier.ujf-grenoble.fr/~parisse/english.html

9.4.3. GTybalt

GTybalt is a free computer algebra system which is built on top of GiNaC, CLN and a program to interpret C and C++ commands. For more information, see

http://www.fis.unipr.it/~stefanw/gtybalt.html

9.4.4. Macaulay 2

Macaulay 2 is a new software system devoted to supporting research in algebraic geometry and commutative algebra. The software is available now in source code for porting, and in compiled form for Linux, Sun OS, Solaris, Windows, and a few other unix machines. You can get it from

http://www.math.uiuc.edu/Macaulay2

9.4.5. Maxima

Maxima is not alone one of the oldest and best computer algebra systems around, it is also one of the only general purpose systems for which there is a free implementation. You can get it from

http://www.ma.utexas.edu/users/wfs/maxima.html

The supported version is GCL-based Maxima 5.6. For CLisp-based Maxima 5.6, edit your `tm_maxima` and replace `-load` by `-i`. For Maxima 5.9-pre, replace `-load` by `-p`. Known problems:

- If you press `return` when a statement is not complete (typically, terminated by `;` or `$`), the interface will hang.
- If you cause the Lisp break prompt to appear, the interface will hang.
- The command `info` is not supported (it is defined in the underlying Lisp, and difficult to support portably).
- Some commands in the debugger work, but some (including `:c`) don't work, nobody knows why.
- The command `load` sometimes behaves mysteriously.

9.4.6. Pari

Pari is a software package for computer-aided number theory. It consists of a C library, `libpari` (with optional assembler cores for some popular architectures), and of the programmable interactive `gp` calculator. You can download Pari from

ftp://megrez.math.u-bordeaux.fr/pub/pari

You will need a version newer than Pari-2.1.0 for use from inside TeXmacs (for an already installed Pari-system, type `gp --version`).

9.4.7. Qcl

QCL is a high level, architecture independent programming language for quantum computers, with a syntax derived from classical procedural languages like C or PASCAL. This allows for the complete implementation and simulation of quantum algorithms (including classical components) in one consistent formalism. The TeXmacs interface is mainly useful for displaying quantum states in a readable way. For more information, see

`http://tph.tuwien.ac.at/~oemer/qcl.html`

Starting from 1.0.0.8, TeXmacs supports QCL 0.4.3 or newer. Users of older versions should upgrade.

9.4.8. Yacas

YACAS is, as it's name suggest, yet another computer algebra system. Things implemented include: arbitrary precision, rational numeric, vector, complex, and matrix computations (including inverses and determinants and solving matrix equations), derivatives, solving, Taylor series, numerical solving (Newtons method), and a lot more non-mathematical algorithms. The language natively supports variables and user-defined functions. There is basic support for univariate polynomials, integrating functions and tensor calculations. You can get YACAS at

`http://www.xs4all.nl/~apinkus/yacas.html`

Chapter 10
Writing TeXmacs style files

One of the fundamental strengths of TeXmacs is the possibility to write your own style files and packages. The purpose of style files is multiple:

- They allow the abstraction of repetitive elements in texts, like sections, theorems, enumerations, etc.
- They form a mechanism which allow you to structure your text. For instance, you may indicate that a given portion of your text is an abbreviation, a quotation or "important".
- Standard document styles enable you to write professionally looking documents, because the corresponding style files have been written with a lot of care by people who know a lot about typography and aesthetics.

The user may select a major style from the Document → Style menu. The major style usually reflects the kind of document you want to produce (like a letter, an article or a book) or a particular layout policy (like publishing an article in a given journal).

Style packages, which are selected from the Document → Style menu, are used for further customization of the major style. For instance, the `number-europe` package enables European-style theorem numbering and the `maxima` package contains macros for customizing the layout of sessions of the Maxima computer algebra system. Several packages may be used together.

When you want to add your own markup to TeXmacs or personalize the layout, then you have to choose between writing a principal style file or a style package. In most cases, you will probably prefer to write a style package, since this will allow you to combine it arbitrary other styles. However, in some cases you may prefer to create a new principal style, usually by personalizing an existing style. This is usually the case if you want to mimic the layout policy of some journal. In this chapter, we will both explain how to write your own style packages and how to customize the standard styles.

10.1. Writing a simple style package

Let us explain on an example how to write a simple style package. First of all, you have to create a new buffer using File → New and select the `source` document style using Document → Style → source. Now save your empty style package in your personal style package directory

```
$HOME/.TeXmacs/packages
```

Notice that the button Texts in the file browser corresponds to the directory

```
$HOME/.TeXmacs/texts
```

Consequently, you can go to the style package directory from there, by double clicking on `..` and next on `packages`. Similarly, the directory

`$HOME/.TeXmacs/styles`

contains your personal style files. After saving your empty style package, it should automatically appear in the Document → Package menu. Notice that style files must be saved using the `.ts` file extension. If you save the style file in a subdirectory of `$HOME/.TeXmacs/packages`, then it will automatically appear in the corresponding submenu of Document → Package.

Let us now create a simple macro hi which displays "Hello world". First type `A-=`, so as to create an assignment. You should see something like

⟨assign||⟩

Now enter "hi" as the first argument and type `A-m` inside the second argument in order to create a macro. You should now see something like

⟨assign|*hi*|⟨macro|⟩⟩

Finally, type the text "Hello world" in the body of the macro. Your document should now consist of the following line:

⟨assign|*hi*|⟨macro|Hello world⟩⟩

After saving your style package, opening a new document and selecting your package in the Document → Use package menu, you may now use the macro hi in your document by typing `\ h i` and hitting `return`.

In a similar way, you may create macros with arguments. For instance, assume that we started entering a macro hello in a similar way as above. Instead of typing "Hello world", we first type `A-left` inside the macro body so as to create an additional argument on the left hand side of the cursor. We next enter the name of the argument, say "name". You should now see something like below:

⟨assign|*hello*|⟨macro|*name*|⟩⟩

In the second argument of the body, we now type "Hello ", `A-#`, "name", `right` and ", how are you today?". After this you should see

⟨assign|*hello*|⟨macro|*name*|Hello *name*, how are you today?⟩⟩

The `A-#` shortcut is used to retrieve the macro argument *name*. Instead of typing `A-#`, "name" and `right`, you may also use the hybrid `\`-key and type `\ n a m e` followed by `return`. After saving your style package, you may again use the macro in any document which uses your package by typing `\ h e l l o` and hitting `return`.

From the internal point of view, all macro definitions are stored in the environment of the TeXmacs typesetter. Besides macros, the environment also contains normal environment variables, such as section counters or the font size. The environment variables can either be globally changed using the assign primitive, or locally, using the with primitive. For instance, when including the line

⟨assign|*section-nr*|-1⟩

in your package, and using `article` as your major style, then the first section will be numbered 0. Similarly, the variant

⟨assign|*hello*|⟨macro|*name*|Hello ⟨with|*font-shape*|small-caps|*name*⟩!⟩⟩

of the hello macro displays the name of the person in SMALL CAPITALS. Notice that the with primitive can also be used to locally redefine a macro. This is for instance used in the definitions of the standard list environments, where the macro which renders list icons is changed inside the body of the list. Yet another variant of the hello macro relies on the standard person macro:

⟨assign|*hello*|⟨macro|*name*|Hello ⟨person|*name*⟩!⟩⟩

In order to produce the macro application ⟨person | *name*⟩, you first have to start a compound tag using `A-c`, type the name "person", insert an argument `A-right`, and enter the argument *name* as before. When you are done, you may press `return` in order to change the compound tag into a person tag. Alternatively, you may type `\`, "person", `A-right` and "name".

By combining the above constructs, an ordinary user should already be able to produce style packages for all frequently used notations. An interesting technique for writing macros which involve complex formulas with some subformulas which may change goes as follows:

1. Type the formula, say $(a_1, \ldots, a_n)$, in an ordinary document.

2. Create the skeleton of your macro in your style package:

   ⟨assign|*n-tuple*|⟨macro|*a*|⟩⟩

3. Copy the formula and paste it into the body of your macro:

   ⟨assign|*n-tuple*|⟨macro|*a*|(a⟨rsub|1⟩,<ldots>,a⟨rsub|n⟩)⟩⟩

4. Replace the subformulas you want to parameterize by macro arguments:

   ⟨assign|*n-tuple*|⟨macro|*a*|(*a*⟨rsub|1⟩,<ldots>,*a*⟨rsub|n⟩)⟩⟩

5. You may now use the macro in documents which use your package:

$$(a_1, \ldots, a_n) = (b_1, \ldots, b_n).$$

10.2. RENDERING OF STYLE FILES AND PACKAGES

10.2.1. ASCII-based or tree-based editing: an intricate choice

Most users are used to edit source code using a conventional editor like EMACS, while presenting the source code in ASCII format. Since all TEXMACS documents are stored as trees, an interesting but complicated question is which format is most suitable for editing such documents. One option is to represent the tree using an ASCII-based format, such as XML, Scheme, or the native format for storing files on a disk. The other option is to edit the trees as such, making no fundamental distinction between source code and normal documents.

In TeXmacs we have chosen to implement the second option. More precisely, any document can be edited in "source mode", which is merely a mode for rendering the document in a way which makes its tree structure particularly apparent. It may be instructive to take an arbitrary document of yours and to take a look at it in "source mode" by enabling Document → View → Edit source tree.

The choice between ASCII-based editing and tree-based editing is non-trivial, because TeXmacs style files and packages have a double nature: they may be seen as programs which specify how to render macros, but these programs naturally contain ordinary content. There are several reasons why users often prefer to edit source code in an ASCII-based format:

1. It is easy to manually format the code so as to make it more readable.

2. In particular, it is easy to add comments.

3. Standard editors like EMACS provide tools for automatic highlighting, indentation, etc.

4. One is not constraint by any "structure" during the editing phase.

Our approach is to reproduce as much of the above advantages in a structured document environment. Although point 4 will obviously be hard to meet when following this approach, we believe that the first three advantages might actually become greater in a structured environment. However, this requires a more profound understanding of how users format and edit source code.

For instance, consider a piece of manually formatted code like

```
if (cond) hop   = 2;
else      holala= 3;
```

Clearly, the user had a particular formatting policy when writing this code. However, this policy does not appear in the document: manual intervention will be necessary if the variable `cond` is renamed `c`, or if the variable `holala` is renamed `hola`.

At the moment, TeXmacs provides no tools for dealing with the above example in an automatic way, but a few tools are already provided. For instance, the user is given a great amount of control on how to indent source code and reasonable defaults are provided as a function of the structure. We also provide high level environments for comments and structured highlighting. Further tools will be developed later and we are open for any suggestions from our users.

10.2.2. Global presentation

In the Source tags group of the Document → View menu, you find several ways to customize the rendering of source trees in your document. We recommend you to play around with the different possibilities in a document of your own (after enabling Document → View → Source tree) or a standard style package in `$TEXMACS_PATH/packages`.

First of all, you may choose between the different major styles "angular", "scheme", "functional" and "LaTeX" for rendering source trees, as illustrated in the figure below:

Angular

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body⟩⟩⟩
```

Scheme

```
(assign "quick-theorem"
    (macro "body"
        (surround (no-indent)"Theorem. " ""
            (arg "body"))))
```

Functional

```
assign (quick-theorem,
    macro (body,
        surround (no-indentTheorem. , ,
            body)))
```

LaTeX

```
assign{quick-theorem}{
    macro{body}{
        surround{no-indentTheorem. }{}{
            body}}}
```

Figure 10.1. Different styles for rendering the same source tree.

Secondly, you may wish to reserve a special treatment to certain tags like concat and document. In the menu Document → View → Special you may specify to which extent you want to treat such tags in a special way:

None. No tags receive a special treatment.

Formatting. Only the formatting tags concat and document are represented as usual.

Normal. In addition to the formatting tags, a few other tags like compound, value and arg are represented in a special way.

Maximal. At the moment, this option is not yet implemented. The intention is to allow the user to write his own customizations and to allow for special rendering of basic operations like plus.

These different options are illustrated below:

None

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨document|
            ⟨surround|⟨concat|⟨no-indent⟩|The-
            orem. ⟩||
                ⟨arg|body⟩⟩⟩⟩⟩
```

Formatting

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            ⟨arg|body⟩⟩⟩⟩
```

Normal

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body⟩⟩⟩
```

Maximal

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body⟩⟩⟩
```

Figure 10.2. Different ways to render special tags.

Another thing which may be controlled by the user is whether the presentation of tags should be compact or stretched out across several lines. Several levels of compactification may be specified in the Document → View → Compactification menu:

Minimal. The tags are all stretched out across several lines.

Only inline tags. All non-inline tags are stretched out across several lines.

Normal. All inline arguments at the start of the tag are represented in a compact way. As soon as we encounter a block argument, the remainder of the arguments are stretched out across several lines.

Inline arguments. All inline arguments are represented in a compact way and only block tags are stretched out across several lines.

Maximal. All source code is represented in a compact way.

The "normal" and "inline arguments" options rarely differ. The visual effect of the different options is illustrated below:

Minimal

```
⟨assign|
    quick-theorem|
    ⟨macro|
        body|
        ⟨surround|
            ⟨concat|
                ⟨no-indent⟩|
                Theorem. ⟩|
            |
            body⟩⟩⟩
```

Only inline tags

```
⟨assign|
    quick-theorem|
    ⟨macro|
        body|
        ⟨surround|
            ⟨no-indent⟩Theorem. |
            |
            body⟩⟩⟩
```

Normal

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body⟩⟩⟩
```

Maximal

```
⟨assign|quick-theorem|⟨macro|body|⟨document|
⟨surround|⟨no-indent⟩Theorem. ||body⟩⟩⟩⟩
```

Figure 10.3. Different levels of compactification.

Finally, the user may specify the way closing tags should be rendered when the tag is stretched out across several lines. The rendering may either be minimalistic, compact, long, or recall the matching opening tag. The different options are illustrated below:

Minimal

```
assign quick-theorem
    macro body
        surround ⟨no-indent⟩Theorem. |
            body
```

Compact

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body⟩⟩⟩
```

Stretched

```
⟨assign|quick-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩Theorem. ||
            body
        ⟩
    ⟩
⟩
```

Repeat

```
⟨\assign|quick-theorem⟩
    ⟨\macro|body⟩
        ⟨\surround|⟨no-indent⟩Theorem. |⟩
            body
        ⟨/surround⟩
    ⟨/macro⟩
⟨/assign⟩
```

Figure 10.4. Different ways to render closing tags.

10.2.3. Local customization

Even though $\mathrm{T_EX_{MACS}}$ tries hard to render source code in a nice way following the global rendering options that you specified, the readability of the source code often needs to be further enhanced locally. In source mode, this can be done using the menus Source → Activation and Source → Presentation. Any local hints on how to render source code are automatically removed from the document when it is being used as a style file or package.

First of all, for certain pieces of content the user may prefer to see them in their "activated" form instead as dead source code. This may for instance be the case for embedded images, or for mathematical symbols, like in

$$\langle \text{assign} | R | \langle \text{macro} | \mathbb{R} \rangle \rangle$$

Such an active presentation may also be preferred for certain more complex macros:

$$\langle \text{assign} | \textit{diag} | \langle \text{macro} | \textit{var} | \textit{dim} | \begin{pmatrix} \textit{var}_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \textit{var}_{\textit{dim}} \end{pmatrix} \rangle \rangle$$

A piece of code can be activated by selecting it and using Source → Activation → Activate or `M-+`. Similarly, a piece of content may be deactivated using `M--` (we used this in the second example above for the rendering of the arguments *var* and *dim*). Activation and deactivation either apply to the whole tree, or to the root only (e.g. Source → Activation → Activate once).

Another way to customize the rendering is to override some of the global rendering options. This is mainly interesting for controlling more precisely which tags have to be stretched across several lines and which tags have to be represented in a compact fashion. For instance, the concat tag can be used in order to concatenate content, as well as for specifying a block of sequential statements, or a combination of both. For instance, in the piece of code

```
⟨assign|my-section|
    ⟨macro|title|
        ⟨concat|
            ⟨header-hook|title⟩|
            ⟨toc-hook|title⟩|
            ⟨my-section-title|title⟩⟩⟩⟩
```

we have stretched the concat tag along several lines using Source → Presentation → Stretched (notice that this implies the concat tag to appear explicitly, so as to avoid confusion with the document tag). Similarly, if a part of the concatenation were to be displayed as usual, then one may use Source → Presentation → Compact:

```
⟨assign|my-section|
    ⟨macro|title|
        ⟨concat|
            ⟨header-hook|title⟩|
            ⟨toc-hook|title⟩|
            ⟨with|font-series|bold|Section:⟩ title⟩⟩⟩
```

At present, we did not implement a way to mark arguments as inline or block, but we might do this later.

A final way to customize the rendering of source code is to apply an arbitrary macro using Source → Presentation → Apply macro or Source → Presentation → Apply macro once. This macro will be automatically removed when you use your document as a style file or package.

10.3. The style-sheet language

In the section about writing a simple style package we already gave you a first impression about the style-sheet language of TeXmacs. In this section, we will give a more complete survey of the available features. For more detailed descriptions, we refer to the chapter about the TeXmacs primitives.

Most style-sheet primitives can be obtained from the Source menu when you are in source mode. You may also obtain them from the Insert → Macro and Insert → Executable menus when editing usual text. Alternatively, you may use the `A-` and `M-e` prefixes in source mode and the `M-i` and `M-e` prefixes otherwise. Furthermore, we recall that the hybrid `\`-key may be used for creating macro-applications or arguments, depending on the context. Finally, the `A-right` and `A-left` keys are used for inserting arguments.

10.3.1. Assignments

All user defined TeXmacs macros and style variables are stored in the "current typesetting environment". This environment associates a tree value to each string variable. Variables whose values are macros correspond to new primitives. The others are ordinary environment variables. The primitives for operating on the environment are available from Source → Define.

You may permanently change the value of an environment variable using the assign primitive, as in the example

⟨assign|*hi*|⟨macro|Hi there!⟩⟩

You may also locally change the values of one or several environment variables using the with primitive:

⟨with|*font-series*|bold|*color*|red|Bold red text⟩

The value of an environment variable may be retrieved using the value primitive. This may for instance be used in order to increase a counter:

⟨assign|*my-counter*|⟨plus|*my-counter*|1⟩⟩

Finally, you may associate logical properties to environment variables using the drd-props primitive. This is explained in more detail in the section about macro primitives.

10.3.2. Macro expansion

The main interest of the TeXmacs' style-sheet language is the possibility to define macros. These come in three flavours: ordinary macros, macros which take an arbitrary number of arguments and external macros, whose expansion is computed by Scheme or a plug-in. The macro-related primitives are available from the Source → Macro menu. Below, we will only describe the ordinary macros. For more details, we refer to the section about macro primitives.

Ordinary macros are usually defined using

```
<assign|my-macro|<macro|x1|···|xn|body>>
```

After such an assignment, my-macro becomes a new primitive with n arguments, which may be called using

```
<my-macro|y1|···|yn>
```

Inside the body of the macro, the arg primitive may be used to retrieve the values of the arguments to the macro.

```
<assign|hello|<macro|name|Hello name, you look nice today!>>
```

It is possible to call a macro with less or more arguments than the expected number. Superfluous arguments are simply ignored. Missing arguments take the nullary uninit primitive as value:

```
<assign|hey|
  <macro|first|second|
    <if|
      <equal|second|?>|
      Hey first, you look lonely today...|
      Hey first and second, you form a nice couple!>>>
```

We finally notice that you are allowed to compute with macros, in a similar way as in functional programming, except that our macros are not closures (yet). For instance:

```
<assign|my-macro-copy|my-macro>
```

The compound tag may be used to apply macros which are the result of a computation:

```
<assign|overloaded-hi|
  <macro|name|
    <compound|
      <if|<nice-weather>|happy-hi|sad-hi>|
      name>>>
```

10.3.3. Formatting primitives

This section contains some important notes on formatting primitives which are not really part of the style-sheet language, but nevertheless very related.

First of all, most TeXmacs presentation tags can be divided in two main categories: inline tags and block tags. For instance, frac is a typical inline tag, whereas theorem is a typical block tag. Some tags, like strong are inline if their argument is and block in the contrary case. When writing macros, it is important to be aware of the inline or block nature of tags, because block tags inside a horizontal concatenation are not rendered in an adequate way. If you need to surround a block tag with some inline text, then you need the surround primitive:

```
⟨assign|my-theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩⟨with|font-series|bold|Theorem. ⟩|⟨right-flush⟩|
            body⟩⟩⟩
```

In this example, we surrounded the body of the theorem with the bold text "Theorem." at the left hand side and a "right-flush" at the right-hand side. Flushing to the right is important in order to make the blue visual border hints look nice when you are inside the environment.

In most cases, $\mathrm{T_EX_{MACS}}$ does a good job in determining which tags are inline and which ones are not. However, you sometimes may wish to force a tag to be a block environment. For instance, the tag very-important defined by

```
⟨assign|very-important|⟨macro|body|⟨with|font-series|bold|color|red|body⟩⟩⟩
```

may both be used as an inline tag and a block environment. When placing your cursor just before the with-tag and hitting `return` followed by `backspace`, you obtain

```
⟨assign|very-important|
    ⟨macro|body|
        ⟨with|font-series|bold|color|red|body⟩⟩⟩
```

Since the body of the macro is now a block, your tag very-important will automatically become a block environment too. In the future, the drd-props primitive will give you even more control over which tags and arguments are inline and which ones are block.

Another important property of tags is whether they contain normal textual content or tabular content. For instance, consider the definition of the standard eqnarray* tag (with a bit of presentation markup suppressed):

```
⟨assign|eqnarray*|
    ⟨macro|body|
        ⟨with|par-mode|center|mode|math|math-display|true|par-sep|0.45fn|
            ⟨surround | ⟨no-page-break*⟩⟨vspace* | 0.5fn⟩ | ⟨vspace | 0.5fn⟩⟨no-
            indent*⟩|
                ⟨tformat|
                    ⟨twith|table-hyphen|y⟩|
                    ⟨twith|table-width|1par⟩|
                    ⟨twith|table-min-cols|3⟩|
                    ⟨twith|table-max-cols|3⟩|
                    ⟨cwith|1|-1|1|1|cell-hpart|1⟩|
                    ⟨cwith|1|-1|-1|-1|cell-hpart|1⟩|
                    body⟩⟩⟩⟩⟩
```

The use of surround indicates that eqnarray* is a block environment and the use of tformat specifies that it is also a tabular environment. Moreover, the twith and cwith are used to specify further formatting information: since we are a block environment, we enable hyphenation and let the table span over the whole paragraph (unused space being equally distributed over the first and last columns). Furthermore, we have specified that the table contains exactly three columns.

Finally, it is important to bear in mind that style-sheets do not merely specify the final presentation of a document, but that they may also contain information for the authoring phase. Above, we have already mentioned the use of the right-flush tag in order to improve the rendering of visual border hints. Similarly, visual hints on invisible arguments may be given in the form of flags:

```
⟨assign|labeled-theorem|
    ⟨macro|id|body|
        ⟨surround|
            ⟨concat|
                ⟨no-indent⟩|
                ⟨flag|Id: id|blue|id⟩|
                ⟨with|font-series|bold|Theorem. ⟩⟩|
            ⟨right-flush⟩|
            body⟩⟩⟩
```

More generally, the specific tag with first argument "screen" may be used to display visual hints, which are removed when printing the document.

10.3.4. Evaluation control

The Source → Evaluation menu contains several primitives to control the way expressions in the style-sheet language are evaluated. The most frequent use of these primitives is when you want to write a "meta-macro" like new-theorem which is used for defining or computing on other macros. For instance:

```
⟨assign|new-theorem|
    ⟨macro|name|text|
        ⟨quasi|
            ⟨assign|⟨unquote|name⟩|
                ⟨macro|body|
                    ⟨surround | ⟨no-indent⟩⟨strong | ⟨unquote | text⟩. ⟩ | ⟨right-
                    flush⟩|
                        body⟩⟩⟩⟩⟩⟩
```

When calling ⟨new-theorem|theorem|Theorem⟩ in this example, we first evaluate all unquote instructions inside the quasi primitive, which yields the expression

```
⟨assign|theorem|
    ⟨macro|body|
        ⟨surround|⟨no-indent⟩⟨strong|Theorem. ⟩|⟨right-flush⟩|
            body⟩⟩⟩
```

Next, this expression is evaluated, thereby defining a macro theorem.

It should be noticed that the TeXmacs conventions for evaluation are slightly different then those from conventional functional languages like Scheme. The subtle differences are motivated by our objective to make it as easy as possible for the user to write macros for typesetting purposes.

For instance, when TeXmacs calls a macro ⟨macro|x_1|⋯|x_n|body⟩ with arguments y_1 until y_n, the argument variables x_1 until x_n are bound to the unevaluated expressions y_1 until y_n, and the body is evaluated with these bindings. The evaluation of y_i takes place each time we request for the argument x_i. In particular, when applying the macro ⟨macro|x|x and again x⟩ to an expression y, the expression y is evaluated twice.

In SCHEME, the bodies of SCHEME macros are evaluated twice, whereas the arguments of functions are evaluated. On the other hand, when retrieving a variable (whether it is an argument or an environment variable), the value is not evaluated. Consequently, a TeXmacs macro

```
⟨assign|foo|⟨macro|x|⟨blah|x|x⟩⟩⟩
```

would correspond to a SCHEME macro

```
(define-macro (foo x)
  `(let ((x (lambda () ,x)))
     (blah (x) (x))))
```

Conversely, the SCHEME macro and function

```
(define-macro (foo x) (blah x x))
(define (fun x) (blah x x))
```

admit the following analogues in TeXmacs:

```
⟨assign|foo|⟨macro|x|⟨eval|⟨blah|⟨quote-arg|x⟩|⟨quote-arg|x⟩⟩⟩⟩⟩
⟨assign|fun|⟨macro|x|⟨with|x*|x|⟨blah|⟨quote-value|x*⟩|⟨quote-value|x*⟩⟩⟩⟩⟩
```

Here the primitives quote-arg and quote-value are used to retrieve the value of an argument resp. an environment variable. The TeXmacs primitives eval, quote, quasiquote and unquote behave in the same way as their SCHEME analogues. The quasi primitive is a shortcut for quasi-quotation followed by evaluation.

10.3.5. Flow control

Besides sequences of instructions, which can be achieved using the concat primitive, and the mechanism of macro expansion, the TeXmacs style-sheet language provides a few other primitive for affecting the control flow: if, case, while and for-each. These primitives are available from the Source → Flow control menu. However, we have to warn the user that the conditional constructs are quite fragile: they only apply to inline content and the accessibility of macro arguments should not to much depend on the conditions.

The most important primitive if, which can be entered using A-?, allows for basic conditional typesetting:

```
⟨assign|appendix|
    ⟨macro|title|body|
        ⟨compound|
            ⟨if|⟨long-document⟩|chapter-appendix|section-appendix⟩|
            title|
            body⟩⟩⟩
```

In this example, appendix is a block environment consisting of a title and a body, and which is rendered as a chapter for long documents and as a section for short ones. Notice that the following implementation would have been incorrect, since the if primitive currently only works for inline content:

```
⟨assign|appendix|
    ⟨macro|title|body|
        ⟨if|
            ⟨long-document⟩|
            ⟨chapter-appendix|title|body⟩|
            ⟨section-appendix|title|body⟩⟩⟩⟩
```

The if primitive may also be used in order to implement optional arguments:

```
⟨assign|hey|
    ⟨macro|first|second|
        ⟨if|
            ⟨equal|second|?⟩|
            Hey first, you look lonely today...|
            Hey first and second, you form a nice couple!⟩⟩⟩
```

However, TeXmacs is not clever enough to detect which arguments are optional and which arguments are accessible (i.e. which arguments can be edited by the user). Therefore, you will have to manually give this information using the drd-props primitive. The case, while and for-each primitives are explained in more detail in the corresponding section on the TeXmacs primitives.

10.3.6. Computational markup

In the menus Source → Arithmetic, Source → Text, Source → Tuple and Source → Condition you will find different primitives for computing with integers, strings, tuples and boolean values. For instance, in the following code, the new-important tag defines a new "important tag" as well as a variant in red:

```
⟨assign|new-important|
    ⟨macro|name|
        ⟨quasi|
            ⟨concat|
                ⟨assign|
                    ⟨unquote|name⟩|
                    ⟨macro|x|⟨with|font-series|bold|x⟩⟩⟩|
                ⟨assign|
                    ⟨unquote|⟨merge|name|-red⟩⟩|
                    ⟨macro|x|⟨with|font-series|bold|color|red|x⟩⟩⟩⟩⟩⟩⟩
```

Here we use the merge primitive in order to concatenate two strings. The different computational primitives are described in more detail in the corresponding section on the TeXmacs primitives.

10.4. Customizing the standard TeXmacs styles

Whenever the standard TeXmacs style files are inadequate for a given purpose, it is possible to write your own style files. However, designing your own style files from scratch may be a complex task, so it is usually preferable to customize the existing styles. This requires some understanding of the global architecture of the standard style files and a more precise understanding of the parts you wish to customize. In this section, we will explain the general principles. For more details, we refer to the chapter on the principal TeXmacs tags.

10.4.1. Organization of style files and packages

Each standard TeXmacs style file or package is based on a potentially finite number of subpackages. From an abstract point of view, this organization may be represented by a labeled tree. For instance, the tree which corresponds to the `article` style is represented below:

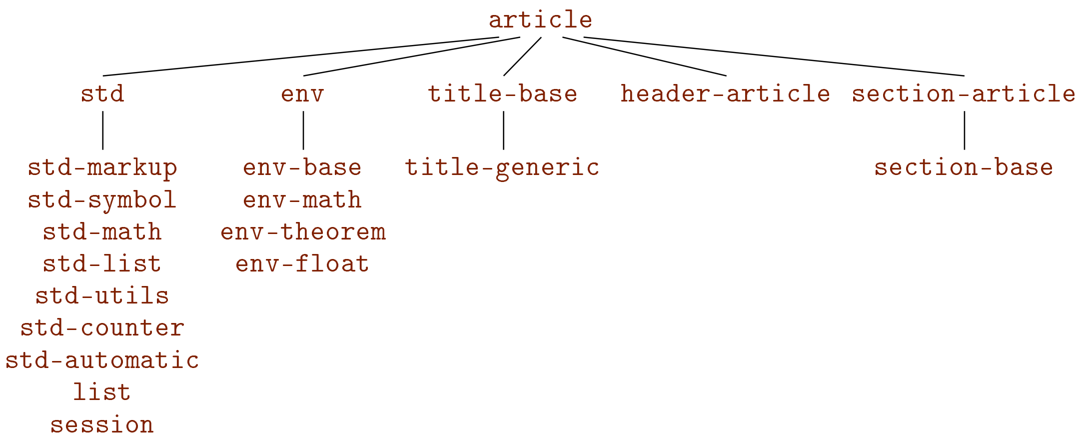


Figure 10.5. The tree with the packages from which the `article` style has been built up. In order to save space, we have regrouped the numerous children of `std` and `env` in vertical lists.

Most of the style packages correspond to a d.t.d. (data type definition) which contains the "abstract interface" of the package, i.e. the exported tags. For instance, the package `std-markup` corresponds to the d.t.d. `std-markup`. Sometimes however, several style packages match the same d.t.d.. For instance, both `header-article` and `header-book` match the d.t.d. `header`, since they merely implement different ways to render the same tags.

When building your own style files or packages, you may use the use-package primitive in order to include other packages. For instance, the `article` style essentially consists of the line

⟨use-package|std|env|title-generic|header-article|section-article⟩

More precisely, the use-package package sequentially includes the style packages corresponding to its arguments. The packages should be in `$TEXMACS_PACKAGE_PATH`, which contains `.`, `~/.TeXmacs/packages` and `$TEXMACS_PATH/packages` by default. Furthermore rendering information for the source code like style-with tags are discarded before evaluation of the files.

REMARK 10.1. We strongly recommend the user to take a look at some of the standard style files and packages which can be found in

```
$TEXMACS_PATH/styles
$TEXMACS_PATH/packages
```

When loading using `C-x C-f`, these paths are in the standard load path. For instance, if you want to take a look at the `std-markup` package, then it suffices to type `C-x C-f`, followed by the file name `std-markup.ts` and `return`.

Remark 10.2. It is also possible to customize the presentation of the source code of the style files and packages themselves, by using other packages in addition to `source` or by using another major style file based on `source`. In that case, the extra markup provided by such packages may be used for presentation purposes of the source code, but it is not exported when using your package in another file.

10.4.2. General principles for customization

Style files and packages basically enrich the current typesetting environment with a combination of

- Environment variables.
- Tags for the end-user.
- Customizable macros.

Furthermore, they may define some tags for intern implementation purposes, which will not be documented in this manual. They may also specify some logical properties of tags using the drd-props primitive.

Environment variables are almost always attributes for controlling the rendering of content, or counters for sections, equations, etc.. Although several simple tags for the end-user like strong may be redefined in your own style files, this practice is not recommended for more complex tags like section. Indeed, the section tag involves many things like resetting subcounters, entering the title into the table of contents and so on. Therefore, special additional macros are provided the customization of such tags, like section-title, section-clean and section-toc.

10.4.3. Customizing the general layout

The general layout of a document is mainly modified by setting the appropriate environment variables for page layout and paragraph layout. For instance, by including the following lines in your style file, you can set the page size to `letter` and the left and right margins to `2in`:

```
⟨assign|page-type|letter⟩
⟨assign|page-odd|2in⟩
⟨assign|page-even|2in⟩
⟨assign|page-right|2in⟩
```

It should be noticed that the environment variables for page layout are quite different in TeXmacs and TeX/LaTeX. In order to make it easier to adapt LaTeX style files to TeXmacs, we have therefore provided the `std-latex` package, which emulates the environment variables from TeX/LaTeX. Typically, this allows you determine the global layout by lines like

```
⟨assign|tex-odd-side-margin|⟨macro|20pt⟩⟩
⟨assign|tex-even-side-margin|⟨macro|20pt⟩⟩
⟨assign|tex-text-width|⟨macro|33pc⟩⟩
```

We notice that macros which return lengths are considered as lengths themselves. In the case of the TEX/LATEX emulation package, we actually *require* all lengths to be macros.

The page headers and footers are usually not determined by global environment variables or macros, since they may change when a new chapter or section is started. Instead, TEXMACS provides the call-back macros header-title, header-author, header-primary and header-secondary. These macros are called when the document title or author are specified or when a new primary or secondary section is started (primary sections are typically chapters in books, or sections in articles). For instance, the following redefinition makes the principal section name appear on even pages, together with the current page number and a wide underline.

```
⟨assign|header-primary|
  ⟨macro|title|nr|type|
    ⟨assign|page-even-header|
      ⟨quasiquote|
        ⟨wide-std-underlined|
          ⟨concat|
            ⟨page-the-page⟩|
            ⟨htab|5mm⟩|
            ⟨unquote|title⟩⟩⟩⟩⟩⟩⟩
```

10.4.4. Customizing list environments

Lists are made up of two principal ingredients: the outer list environment and the inner items. List environments may either be customized by customizing or redefining the rendering macros for these environments, or defining additional list environments which match the same abstract interface.

The rendering of the outer list environment is controlled by the render-list macro which takes the body of the list as its argument. For instance, consider the following redefinition of render-list:

```
⟨assign|render-list|
  ⟨macro|body|
    ⟨surround|
      ⟨no-page-break*⟩⟨vspace*|0.5fn⟩|
      ⟨right-flush⟩⟨vspace|0.5fn⟩⟨no-indent*⟩|
      ⟨with|par-left|⟨plus|par-left|3fn⟩|par-right|⟨plus|par-right|3fn⟩|
      body⟩⟩⟩⟩
```

This redefinition affects the rendering of all list environments (itemize, enumerate, etc.) by reducing the right margin with a length of `3fn`:

- This text, which has been made so long that it does not fit on a single line, is indented on the right hand side by `3fn`.

    1. This text is indented by an additional `3fn` on the right hand side, since it occurs inside a second list environment.

- Once again: this text, which has been made so long that it does not fit on a single line, is indented on the right hand side by `3fn`.

In a similar way, you may customize the rendering of list items by redefining the macros aligned-item and compact-item. These macros both take one argument with the text of the item and render it either in a right-aligned way (such that subsequent text is left aligned) or in a left-aligned way (such that subsequent text may not be aligned). For instance, consider the following redefinition of aligned-item:

```
⟨assign|aligned-item|
  ⟨macro|x|
    ⟨concat|
      ⟨vspace*|0.5fn⟩|
      ⟨with|par-first|-3fn|⟨yes-indent⟩⟩|
      ⟨resize|⟨with|color|red|x⟩|r-2.5fn||r+0.5fn|⟩⟩⟩⟩
```

Then items inside all list environments with compact items will appear in red:

- This list and aligned descriptions have red items.

  C1. First condition.

  C2. Second condition.

- The items of compact description lists are rendered using compact-item.

  Gnus and gnats. Nice beasts.

  Micros and softies. Evil beings.

REMARK 10.3. The macros aligned-item and compact-item are required to produce inline content, so that they may be used in order to surround blocks. In particular, several other internal macros (aligned-space-item, long-compact-strong-dot-item, etc.) are based on aligned-item and compact-item, and used for the rendering of the different types of lists (itemize-arrow, description-long, etc.). In the future, we also plan to extend item and item* with a compulsory *body* argument. When customizing the list environments, it is important to keep that in mind, so as to make your style-sheets upward compatible.

The `std-list` d.t.d. also provides a macro new-list to define new lists. Its syntax is ⟨new-list|*name*|*item-render*|*item-transform*⟩, where *name* is the name of the new list environment, *item-render* an (inline) macro for rendering the item and *item-transform* an additional transformation which is applied on the item text. For instance, the enumerate-roman environment is defined by

```
⟨new-list|enumerate-roman|aligned-dot-item|⟨macro|x|⟨number|x|roman⟩⟩⟩
```

10.4.5. Customizing numbered textual environments

TeXmacs provides three standard types of numbered textual environments: theorem-like environments, remark-like environments and exercise-like environments. The following aspects of these environments can be easily customized:

- Adding new environments.

- Modifying the rendering of the environments.
- Numbering the theorems in a different way.

Defining new environments.

First of all, new environments can be added using the meta-macros new-theorem, new-remark and new-exercise. These environments take two arguments: the name of the environment and the name which is used for its rendering. For instance, you may wish to define the environment experiment by

```
<new-theorem|experiment|Experiment>
```

When available in the $\mathrm{T_EX_{MACS}}$ dictionaries, the text "Experiment" will be automatically translated when your document is written in a foreign language. In the section about how to define new environments, it is also explained how to define other numbered textual environments (besides theorems, remarks and exercises).

Customization of the rendering.

The principal rendering of the environments can be customized by redefining the render-theorem, render-remark and render-exercise macros. These macros take the *name* of the environment (like "Theorem 1.2") and its *body* as arguments. For instance, if you want theorems to appear in a slightly indented way, with a slanted body, then you may redefine render-theorem as follows:

```
<assign|render-theorem|
    <macro|which|body|
        <padded-normal|1fn|1fn|
            <surround|<theorem-name|which<theorem-sep>>||
                <with | font-shape | slanted | par-left | <plus | par-left | 1.5fn> |
                body>>>>>
```

This redefinition produces the following effect:

THEOREM 10.4. *This is a theorem which has been typeset in a slanted font.*

By default, the theorems are rendered as remarks with the only difference that their bodies are typeset in an italic font. Hence, redefining the render-remark macro will also affect the rendering of theorems. The default render-proof macro is also based on render-remark.

Instead of redefining the entire rendering, the user might just wish to customize the way names of theorems are rendered or redefine the separator between the name and the body. As the user may have noticed by examining the above redefinition of render-theorem, these aspects are controlled by the macros theorem-name and theorem-sep. For instance, consider the following redefinitions:

```
<assign|theorem-name|<macro|name|<with|color|dark red|font-series|bold|
name>>>

<assign|theorem-sep|<macro|: >>
```

Then theorem-like environments will be rendered as follows:

Proposition 10.5: *This proposition is rendered in is a fancy way.*

Customization of the numbering.

In the sections about counters and counter groups, it is explained how to customize the counters of numbered environments for particular purposes. For instance, by redefining inc-theorem, you may force theorems to reset the counter of corollaries:

```
⟨quasi|
  ⟨assign|
    inc-theorem|
    ⟨macro|⟨compound|⟨unquote|inc-theorem⟩⟩⟨reset-corollary⟩⟩⟩⟩
```

Notice the trick with quasi and unquote in order to take into account additional action which might have been undertaken by the previous value of the macro inc-theorem.

The following code from `number-long-article.ts` is used in order to prefix all standard environments with the number of the current section:

```
⟨assign|section-clean|⟨macro|⟨reset-subsection⟩⟨reset-std-env⟩⟩⟩
⟨assign|display-std-env|⟨macro|nr|⟨section-prefix⟩nr⟩⟩
```

10.4.6. Customizing sectional tags

By default, TEXMACS provides the standard sectional tags from LATEX part, chapter, section, subsection, subsubsection, paragraph, subparagraph, as well as the special tag appendix. TEXMACS also implements the unnumbered variants part*, chapter*, etc. and special section-like tags bibliography, table-of-contents, the-index, the-glossary, list-of-figures, list-of-tables.

REMARK 10.6. Currently, the sectional tags take one argument, the section title, but a second argument with the body of the section is planned to be inserted in the future (see the experimental `structured-section` package). For this reason (among others), style files should never redefine the main sectional tags, but rather customize special macros which have been provided to this effect.

From a global point of view, an important predicate macro is sectional-short-style. When it evaluates to `true`, then appendices, tables of contents, etc. are considered to be at the same level as sections. In the contrary case, they are at the same level as chapters. Typically, articles use the short sectional style whereas book use the long style.

The rendering of a sectional tag *x* is controlled through the macros *x*-sep, *x*-title and *x*-numbered-title. The *x*-sep macro prints the separator between the section number and the section title. It defaults to the macro sectional-sep, which defaults in its turn to a wide space. For instance, after redefining

```
⟨assign|sectional-sep|⟨macro| − ⟩⟩
```

sectional titles would typically look like

10.1 – Hairy GNUs

The *x*-title and *x*-numbered-title macros respectively specify how to render unnumbered and numbered section titles. Usually, the user only needs to modify *x*-title, since *x*-numbered-title is based on *x*-title. However, if the numbers have to be rendered in a particular way, then it may be necessary to redefine *x*-numbered-title. For instance, consider the redefinition

```
⟨assign|subsection-numbered-title|
  ⟨macro|name|
    ⟨sectional-normal|
      ⟨with|font-series|bold|⟨the-subsection⟩. ⟩name⟩⟩⟩
```

This has the following effect on the rendering of subsection titles:

2.3. Very hairy GNUs

Notice that the `section-base` package provides several useful helper macros like sectional-normal.

Remark 10.7. Sectional titles can either be rendered in a “short” or in the “long” fashion. By default, paragraphs and subparagraphs use the short rendering, for which the body starts immediately at the right of the title:

My paragraph. Blah, blah, and more blahs...

All other sectional tags use the long rendering, in which case the section title takes a separate line on its own:

My section

Blah, blah, and more blahs...

We do not recommend to modify the standard settings (i.e. to render paragraphs in a long way or sections in a short way). If you really want to do so, then we recommend to redefine the corresponding environment variables *enrich-x-long*. This will ensure upward compatibility when sectional tags will take an additional argument (see remark 10.6).

Besides their rendering, several other aspects of sectional tags can be customized:

- The call-back macro *x*-clean can be used for cleaning some counters when a new section is started. For instance, in order to prefix all standard environments by the section counter, you may use the following lines:

  ```
  ⟨assign|section-clean|⟨macro|⟨reset-subsection⟩⟨reset-std-env⟩⟩⟩
  ⟨assign|display-std-env|⟨macro|nr|⟨section-prefix⟩nr⟩⟩
  ```

- The call-back macro *x*-header should be used in order to modify page headers and footers when a new section is started. Typically, this macro should call header-primary, or header-secondary, or do nothing.

- The call-back macro *x-toc* should be used in order to customize the way new sections appear in the table of contents.

10.4.7. Customizing the treatment of title information

TeXmacs uses the doc-data tag in order to specify global data for the document. These data are treated in two stages by the doc-data macro. First, the document data are separated into several categories, according to whether the data should be rendered as a part of the main title or in footnotes or the abstract. Secondly, the data in each category are rendered using suitable macros.

Each child of the doc-data is a tag with some specific information about the document. Currently implemented tags are doc-title, doc-subtitle, doc-author-data, doc-date, doc-running-title, doc-running-author, doc-keywords, doc-AMS-class and doc-note. The doc-author-data tag may occur several times and is used in order to specify data for each of the authors of the document. Each child of the doc-author-data tag is a tag with information about the corresponding author. Currently implemented tags with author information are author-name, author-address, author-email, author-homepage and author-note.

Most of the tags listed above also correspond to macros for rendering the corresponding information as part of the main title. For instance, if the date should appear in bold italic at a distance of at least `1fn` from the other title fields, then you may redefine doc-date as

```
⟨assign|doc-date|
    ⟨macro|body|
        ⟨concat|
            ⟨vspace*|1fn⟩|
            ⟨doc-title-block|⟨with|font-shape|italic|font-series|bold|body⟩⟩|
            ⟨vspace|1fn⟩⟩⟩⟩
```

The title-block macro is used in order to make the text span appropriately over the width of the title. The doc-title and author-name are special in the sense that they also render possible references to footnotes. For this reason, you should rather customize the doc-render-title and author-render-name macros in order to customize the rendering of the title and the name themselves.

Notice also that the doc-running-title and author-running-author macros do not render anything, but rather call the header-title and header-author call-backs for setting the appropriate global page headers and footers. By default, the running title and author are extracted from the usual title and author names.

In addition to the rendering macros which are present in the document, the main title (including author information, the date, etc.) is rendered using the doc-make-title macro. The author information, as part of the main title, is rendered using doc-author or doc-authors, depending on whether the document has one or more authors. Footnotes to the title or to one of the authors are rendered using doc-title-note resp. doc-author-note. These footnote macros always expect a document tag on input, because they may compress it into a horizontal concatenation.

The first stage of processing the document data is more complex and the reader is invited to take a look at the short descriptions of the macros which are involved in this process. It is also good to study the definitions of these macros in the package itself. In order to indicate the way things work, we finish with an example on how the email address and homepage of an author can be rendered in a footnote instead of the main title:

```
⟨assign|doc-author-main|
    ⟨macro|data|
        ⟨quasi|
            ⟨unquote*|⟨select|⟨quote-arg|data⟩|author-name⟩⟩
            ⟨unquote*|⟨select|⟨quote-arg|data⟩|author-address⟩⟩⟩⟩⟩
⟨assign|doc-author-data-note|
    ⟨xmacro|data|
        ⟨quasi|
            ⟨unquote*|⟨select|⟨quote-arg|data⟩|author-email⟩⟩
            ⟨unquote*|⟨select|⟨quote-arg|data⟩|author-homepage⟩⟩
            ⟨unquote*|⟨select|⟨quote-arg|data⟩|author-note|document|⟨pat-
            any⟩⟩⟩⟩⟩⟩
```

10.5. Further notes and tips

10.5.1. Customizing arbitrary tags

Imagine that you want to change the rendering of a given tag, like lemma. As a general rule, TeXmacs provides a set of well-chosen macros which can be customized by the user so as to obtain the desired effect. For instance, as we have seen above, you should use modify one of the macros render-theorem, theorem-name or theorem-sep in order to customize the rendering of lemma and all other theorem-like environments.

However, in some cases, it may not be clear which “well-chosen” macro to customize. If we just wanted to change the presentation of lemmas and not of any other theorem-like environments, then we clearly cannot modify render-theorem, theorem-name or theorem-sep. In other cases, the user may not want to invest his time in completely understanding the macro hierarchy of TeXmacs, and find out about the existence of render-theorem, theorem-name and theorem-sep.

So imagine that you want all lemmas to appear in red. One thing you can always do is copy the original definition of lemmas in a safe place and redefine the lemma macro on top of the original definition:

```
⟨assign|orig-lemma|lemma⟩
⟨assign|lemma|⟨macro|body|⟨with|color|red|⟨orig-lemma|body⟩⟩⟩⟩
```

Alternatively, if only the text inside the lemma should be rendered in red, then you may do:

```
⟨assign|orig-lemma|lemma⟩
⟨assign|lemma|⟨macro|body|⟨orig-lemma|⟨with|color|red|body⟩⟩⟩⟩
```

Of course, you have to be careful that the name orig-lemma is not already in use.

Another frequent situation is that you only want to modify the rendering of a tag when it is used inside another one. On the web, the *Cascading Style Sheet* language (CSS) provides a mechanism for doing this. In TeXmacs, you may simulate this behaviour by redefining macros inside a with. For instance, imagine that we want the inter-paragraph space inside lists inside theorem-like environments to vanish. Then we may use:

```
⟨assign|orig-render-theorem|render-theorem⟩
⟨assign|render-theorem|
   ⟨macro|name|body|
      ⟨with|orig-render-list|render-list|
         ⟨with|render-list|⟨macro|x|⟨with|par-par-sep|0fn|⟨orig-render-list|
         x⟩⟩⟩|
            ⟨orig-render-theorem|
               name|
               body⟩⟩⟩⟩⟩
```

On the one hand side, this mechanism is a bit more complex than CSS, where it suffices to respecify the *par-par-sep* attribute of lists inside theorems. On the other hand, it is also more powerful, since the render-theorem macro applies to all theorem-like environments at once. Furthermore, if the above mechanism is to be used frequently, then real hackers may simplify the notations using further macro magic.

10.5.2. Standard utilities

In the package `std-utils`, the user may find several useful additional macros for writing style files. It mainly contains macros for

- Writing block environments which span over the entire paragraph width. Notice that the `title-base` package provides some additional macros for wide section titles.
- Writing wide block environments which are underlined, overlined or in a frame box.
- Recursive indentation.
- Setting page headers and footers.
- Localization of text.

It is good practice to use these standard macros whenever possible when writing style files. Indeed, the low-level TeXmacs internals may be subject to minor changes. When building upon standard macros with a clear intention, you increase the upward compatibility of your style-sheets.

Chapter 11

Customizing TeXmacs

One major feature of TeXmacs is that it can be highly customized. First of all, the most important aspects of the program can be configured in Edit → Preferences. Most other parts of TeXmacs can be entirely adapted or reprogrammed using the Guile/Scheme extension language. In the sequel, we give a short overview of how this works in simple cases.

11.1. Introduction to the Guile extension language

Like Emacs, TeXmacs comes with a Lisp-like extension language, namely the Guile Scheme dialect from the Gnome project. For documentation about Guile Scheme, we refer to

`http://www.gnu.org/software/guile/guile.html`

Scheme has the advantage that it may be extended with extern C and C++ types and routines. In our case, we have extended Scheme with routines which you can use to create your own menus and key-combinations, and even to write your own extensions to TeXmacs.

If you have downloaded the source files of TeXmacs, then it may be interesting for you to take a look at the files

```
Guile/Glue/build-glue-basic.scm
Guile/Glue/build-glue-editor.scm
Guile/Glue/build-glue-server.scm
```

These three "glue" files contain the C++ routines, which are visible within Scheme. In what follows, we will discuss some of the most important routines. We plan to write a more complete reference guide later. You may also take a look at the scheme `.scm` files in the directory `$TEXMACS_PATH/progs`.

11.2. Writing your own initialization files

When starting up, TeXmacs executes the file

`$TEXMACS_PATH/progs/init-texmacs.scm`

as well as your personal initialization file

`$TEXMACS_HOME_PATH/progs/my-init-texmacs.scm`

if it exists. By default, the path `$TEXMACS_HOME_PATH` equals `.TeXmacs`. Similarly, each time you create a new buffer, the file

`$TEXMACS_PATH/progs/init-buffer.scm`

is executed, as well as

```
$TEXMACS_HOME_PATH/progs/my-init-buffer.scm
```

if it exists.

11.3. Creating your own dynamic menus

You may define (or modify) a (part of a) menu with name `name` using

```
(menu-bind name . prog)
```

and append new entries to an existing (part of a) menu with name `name` using

```
(menu-extend name . prog)
```

Here `prog` is a program which represents the entries of the menu. In particular, you may take a look at the files in the directory

```
$TEXMACS_PATH/progs/menu
```

in order to see how the standard TeXmacs menus are defined.

More precisely, the program `prog` in `menu-set` or `menu-append` is a list of entries of one of the following forms:

```
(=> "pulldown menu name" menu-definition)
(-> "pullright menu name" menu-definition)
("entry" action)
---
(if condition menu-definition)
(link variable)
```

The constructors `=>` and `->` are used to create pulldown or pullright menus and `menu-definition` should contain a program which creates the submenu. The constructor `("entry" action)` creates an ordinary entry, where `action` will be compiled and executed when you click on `entry`. Items of a menu may be separated using `---`. The constructor `if` is used for inserting menu items only if a certain `condition` is satisfied (for instance, if we are in math mode).

Finally, if you declared a menu `name`, then you may use this menu indirectly using the `link` constructor. This indirect way of declaring submenus has two advantages

- An “indirect” submenu may be linked to as many menus as we like.
- New items may be added to “indirect” submenus *a posteriori* using `menu-append`.

The main TeXmacs menus are `texmacs-menu`, `texmacs-popup-menu`, `texmacs-main-icons`, `texmacs-context-icons` and `texmacs-extra-icons`. Other standard indirect menus are `file-menu`, `edit-menu`, `insert-menu`, `text-menu`, `paragraph-menu`, `document-menu`, `options-menu` and `help-menu`.

11.4. Creating your own keyboard shortcuts

Keymaps are specified using the command

```
(kbd-map predicate . keymaps)
```

The predicate specifies under which circumstances the keymaps are valid. Examples of predicates are `always?`, `in-math?` and `in-french?`, but the user may define his own predicates. Each item in `keymaps` is of one of the following forms:

```
(key-combination action_1 ... action_n)
(key-combination result)
(key-combination result help-message)
```

In the first case, the `action_i` are Scheme commands associated to the string `key-combination`. In the second and third case, `result` is a string which is to be inserted in the text when the `key-combination` has been completed. An optional `help-message` may be displayed when the `key-combination` is finished.

11.5. Other interesting files

Some other files may also be worth looking at:

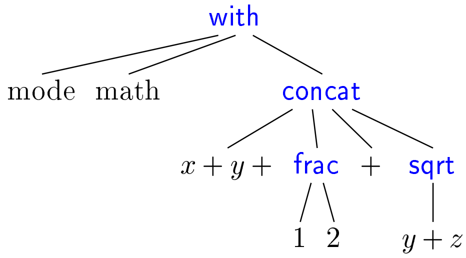
- `$TEXMACS_PATH/fonts/enc` contains encodings for different TeX fonts.
- `$TEXMACS_PATH/fonts/virtual` contains definitions of virtual characters.
- `$TEXMACS_PATH/langs/natural/dic` contains the current dictionaries used by TeXmacs.
- `$TEXMACS_PATH/langs/natural/hyphen` contains hyphenation patterns for various languages.
- `$TEXMACS_PATH/progs/fonts` contains Scheme programs for setting up the fonts.

Chapter 12

The TeXmacs format

12.1. TeXmacs trees

All TeXmacs documents or document fragments can be thought of as *trees*. For instance, the tree



typically represents the formula

$$x+y+\frac{1}{2}+\sqrt{y+z} \tag{12.1}$$

Internal nodes of TeXmacs trees.

Each of the internal nodes of a TeXmacs tree is a string symbol and each of the leafs is an ordinary string. A string symbol is different from a usual string only from the efficiency point of view: TeXmacs represents each symbol by a unique number, so that it is extremely fast to test weather two symbols are equal.

Leafs of TeXmacs trees.

Currently, all strings are represented using the *universal TeXmacs encoding*. This encoding coincides with the Cork font encoding for all characters except "<" and ">". Character sequences starting with "<" and ending with ">" are interpreted as special extension characters. For example, `<alpha>` stands for the letter α. The semantics of characters in the universal TeXmacs encoding does not depend on the context (currently, cyrillic characters are an exception, but this should change soon). In other words, the universal TeXmacs encoding may be seen as an analogue of Unicode. In the future, we might actually switch to Unicode.

The string leafs either contain ordinary text or special data. TeXmacs supports the following atomic data types:

Boolean numbers. Either `true` or `false`.

Integers. Sequences of digits which may be preceded by a minus sign.

Floating point numbers. Specified using the usual scientific notation.

Lengths. Floating point numbers followed by a length unit, like `29.7cm` or `2fn`.

Serialization and preferred syntax for editing.

When storing a document as a file on your harddisk or when copying a document fragment to the clipboard, TeXmacs trees have to be represented as strings. The conversion without loss of information of abstract TeXmacs trees into strings is called *serialization* and the inverse process *parsing*. TeXmacs provides three ways to serialize trees, which correspond to the standard TeXmacs format, the XML format and the SCHEME format.

However, it should be emphasized that the preferred syntax for modifying TeXmacs documents is the screen display inside the editor. If that seems surprising to you, consider that a syntax is a way to represent information in a form suitable to understanding and modification. The on-screen typeset representation of a document, together with its interactive behaviour, is a particularly concrete syntax. Moreover, in the Document → View menu, you may find different ways to customize the way documents are viewed, such as different levels of informative flags and a "source tree" mode for editing style files.

12.2. TeXmacs documents

Whereas TeXmacs document fragments can be general TeXmacs trees, TeXmacs documents are trees of a special form which we will describe now. The root of a TeXmacs document is necessarily a document tag. The children of this tag are necessarily of one of the following forms:

⟨TeXmacs|*version*⟩ (TeXmacs version)

This mandatory tag specifies the version of TeXmacs which was used to save the document.

⟨project|*ref*⟩ (part of a project)

An optional project to which the document belongs.

⟨style|*version*⟩
⟨style|⟨tuple|*style*|*pack-1*|···|*pack-n*⟩⟩ (style and packages)

An optional style and additional packages for the document.

⟨body|*content*⟩ (body of the document)

This mandatory tag specifies the body of your document.

⟨initial|*table*⟩ (initial environment)

Optional specification of the initial environment for the document, with information about the page size, margins, etc.. The *table* is of the form ⟨collection|*binding-1*|···|*binding-n*⟩. Each *binding-i* is of the form ⟨associate|*var-i*|*val-i*⟩ and associates the initial value *val-i* to the environment variable *var-i*. The initial values of environment variables which do not occur in the table are determined by the style file and packages.

⟨references|*table*⟩ (references)

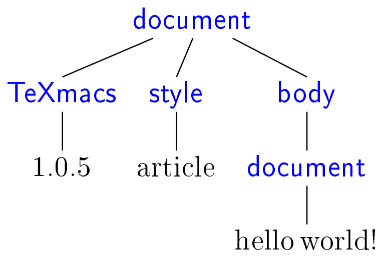
An optional list of all valid references to labels in the document. Even though this information can be automatically recovered by the typesetter, this recovery requires several passes. In order to make the behaviour of the editor more natural when loading files, references are therefore stored along with the document.

The *table* is of a similar form as above. In this case a tuple is associated to each label. This tuple is either of the form ⟨tuple|*content*|*page-nr*⟩ or ⟨tuple|*content*|*page-nr*|*file*⟩. The *content* corresponds to the displayed text when referring to the label, *page-nr* to the corresponding page number, and the optional *file* to the file where the label was defined (this is only used when the file is part of a project).

⟨auxiliary|*table*⟩ (auxiliary data attached to the file)

This optional tag specifies all auxiliary data attached to the document. Usually, such auxiliary data can be recomputed automatically from the document, but such recomputations may be expensive and even require tools which are not necessarily installed on your system. The *table*, which is specified in a similar way as above, associates auxiliary content to a key. Standard keys include `bib`, `toc`, `idx`, `gly`, etc.

Example 12.1. An article with the simple text "hello world!" is represented as



12.3. Default serialization

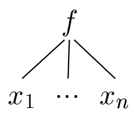
Documents are generally written to disk using the standard TeXmacs syntax (which corresponds to the `.tm` and `.ts` file extensions). This syntax is designed to be unobtrusive and easy to read, so the content of a document can be easily understood from a plain text editor. For instance, the formula (12.1) is represented by

```
<with|mode|math|x+y+<frac|1|2>+<sqrt|y+z>>
```

On the other hand, TeXmacs syntax makes style files difficult to read and is not designed to be hand-edited: whitespace has complex semantics and some internal structures are not obviously presented. Do not edit documents (and especially style files) in the TeXmacs syntax unless you know what you are doing.

Main serialization principle.

The TeXmacs format uses the special characters `<`, `|`, `>`, `\` and `/` in order to serialize trees. By default, a tree like



(12.2)

is serialized as

```
<f|x1|...|xn>
```

If one of the arguments $x_1, \ldots, x_n$ is a multi-paragraph tree (which means in this context that it contains a document tag or a collection tag), then an alternative long form is used for the serialization. If `f` takes only multi-paragraph arguments, then the tree would be serialized as

```
<\f>
  x₁
<|f>
  ...
<|f>
  xₙ
</f>
```

In general, arguments which are not multi-paragraph are serialized using the short form. For instance, if `n=5` and x_3 and x_5 are multi-paragraph, but not x_1, x_2 and x_4, then (12.2) is serialized as

```
<\f|x₁|x₂>
  x₃
<|f|x₄>
  x₅
</f>
```

The escape sequences `\<`, `\|`, `\>` and `\\` may be used to represent the characters `<`, `|`, `>` and `\`. For instance, $\alpha+\beta$ is serialized as `\<alpha\>+\<beta\>`.

Formatting and whitespace.

The document and concat primitives are serialized in a special way. The concat primitive is serialized as usual concatenation. For instance, the text "an *important* note" is serialized as

```
an <em|important> note
```

The document tag is serialized by separating successive paragraphs by double newline characters. For instance, the quotation

> Ik ben de blauwbilgorgel.
>
> Als ik niet wok of worgel,

is serialized as

```
<\quote-env>
  Ik ben de blauwbilgorgel.

  Als ik niet wok of worgel,
</quote-env>
```

Notice that whitespace at the beginning and end of paragraphs is ignored. Inside paragraphs, any amount of whitespace is considered as a single space. Similarly, more than two newline characters are equivalent to two newline characters. For instance, the quotation might have been stored on disk as

```
<\quote-env>
  Ik ben de           blauwbilgorgel.



  Als ik niet wok of          worgel,
</quote-env>
```

The space character may be explicitly represented through the escape sequence "\ ". Empty paragraphs are represented using the escape sequence "\;".

Raw data.

The raw-data primitive is used inside TeXmacs for the representation of binary data, like image files included into the document. Such binary data is serialized as

```
<#binary-data>
```

where the ***binary-data*** is a string of hexadecimal numbers which represents a string of bytes.

12.4. XML SERIALIZATION

For compatability reasons with the XML technology, TeXmacs also supports the serialization of TeXmacs documents in the XML format. However, the XML format is generally more verbose and less readable than the default TeXmacs format. In order to save or load a file in the XML format (using the `.tmml` extension), you may use File → Export → XML resp. File → Import → XML.

It should be noticed that TeXmacs documents do not match a predefined DTD, since the appropriate DTD for a document depends on its style. The XML format therefore merely provides an XML representation for TeXmacs trees. The syntax has both been designed to be close to the tree structure and use conventional XML notations which are well supported by standard tools.

The encoding for strings.

The leafs of TeXmacs trees are traslated from the universal TeXmacs encoding into Unicode. Characters without Unicode equivalents are represented as entities (in the future, we rather plan to create a `tmsym` tag for representing such characters).

XML representation of regular tags.

Trees with a single child are simply represented by the corresponding XML tag. In the case when a tree has several children, then each child is enclosed into a `tm-arg` tag. For instance, $\sqrt{x+y}$ is simply represented as

```
<sqrt>y+z</sqrt>
```

whereas the fraction $\frac{1}{2}$ is represented as

```
<frac>
  <tm-arg>1</tm-arg>
  <tm-arg>2</tm-arg>
</frac>
```

In the above example, the whitespace is ignored. Whitespace may be preserved by setting the standard `xml:space` attribute to `preserve`.

Special tags.

Some tags are represented in a special way in XML. The concat tag is simply represented by a textual concatenation. For instance, $\frac{1}{2}+\sqrt{x+y}$ is represented as

```
<frac><tm-arg>1</tm-arg><tm-arg>2</tm-
arg></frac>+<sqrt>y+z</sqrt>
```

The document tag is not explicitly exported. Instead, each paragraph argument is enclosed within a `tm-par` tag. For instance, the quotation

> Ik ben de blauwbilgorgel.
>
> Als ik niet wok of worgel,

is represented as

```
<quote-env>
  <tm-par>
    Ik ben de blauwbilgorgel.
  </tm-par>
  <tm-par>
    Als ik niet wok of worgel,
  </tm-par>
</quote-env>
```

A with tag with only string attributes and values is represented using the standard XML attribute notation. For instance, "some blue text" would be represented as

```
some <with color="blue">blue</with> text
```

Conversily, TeXmacs provides the attr primitive in order to represent attributes of XML tags. For instance, the XML fragment

```
some <mytag beast="heary">special</mytag> text
```

would be imported as "some ⟨my-tag|⟨attr|beast|heary⟩|special⟩ text". This will make it possible, in principle, to use TeXmacs as an editor of general XML files.

12.5. Scheme serialization

Users may write their own extensions to TeXmacs in the Scheme extension language. In that context, TeXmacs trees are usually represented by Scheme expressions. The Scheme syntax was designed to be predictable, easy to hand-edit, and expose the complete internal structure of the document. For instance, the formula (12.1) is represented by

```
(with "mode" "math" (concat "x+y+" (frac "1" "2") "+" (sqrt
"y+z")))
```

The Scheme representation may also be useful in order to represent complex macros with a lot of programmic content. Finally, Scheme is the safest format when incorporating TeXmacs snippets into emails. Indeed, both the standard TeXmacs format and the XML serialization may be quite sensitive to white-space.

In order to save or load a document in Scheme format, you may use File → Export → Scheme resp. File → Import → Scheme. Files saved in Scheme format can easily be processed by external Scheme programs, in the same way as files saved in XML format can easily be processed by tools for processing XML, like XSLT.

In order to copy a document fragment to an email in Scheme format, you may use Edit → Copy to → Scheme. Similarly, you may paste external Scheme fragments into TeXmacs using Edit → Paste from → Scheme. The Scheme format may also used interactively inside Scheme sessions or interactive commands. For instance, typing `M-x` followed by the interactive command

```
(insert '(frac "1" "2"))
```

inserts the fraction $\frac{1}{2}$ at the current cursor position.

12.6. The typesetting process

In order to unserstand the TeXmacs document format well, it is useful to have a basic understanding about how documents are typeset by the editor. The typesetter mainly rewrites logical TeXmacs trees into physical *boxes*, which can be displayed on the screen or on paper (notice that boxes actually contain more information than is necessary for their rendering, such as information about how to position the cursor inside the box or how to make selections).

The global typesetting process can be subdivided into two major parts (which are currently done at the same stage, but this may change in the future): evaluation of the TeXmacs tree using the stylesheet language, and the actual typesetting.

The typesetting primitives are designed to be very fast and they are built-in into the editor. For instance, one has typesetting primitives for horizontal concatenations (concat), page breaks (page-break), mathematical fractions (frac), hyperlinks (hlink), and so on. The precise rendering of many of the typesetting primitives may be customized through the built-in environment variables. For instance, the environment variable *color* specifies the current color of objects, *par-left* the current left margin of paragraphs, etc.

The stylesheet language allows the user to write new primitives (macros) on top of the built-in primitives. It contains primitives for definining macros, conditional statements, computations, delayed execution, etc. The stylesheet language also provides a special extern tag which offers you the full power of the Scheme extension language in order to write macros.

It should be noticed that user-defined macros have two aspects. On the one hand they usually perform simple rewritings. For instance, the macro

⟨assign|*seq*|⟨macro|*var*|*from*|*to*|*var*$_{from}$, …, *var*$_{to}$⟩⟩

is a shortcut in order to produce sequences like $a_1, \ldots, a_n$. When macros perform simple rewritings like in this example, the children *var*, *from* and *to* of the seq tag remain *accessible* from within the editor. In other words, you can position the cursor inside them and modify them. User defined macros also have a synthetic or computational aspect. For instance, the dots of a seq tag as above cannot be edited by the user. Similarly, the macro

```
⟨assign|square|⟨macro|x|⟨times|x|x⟩⟩⟩
```

serves an exclusively computational purpose. As a general rule, synthetic macros are sometimes easier to write, but the more accessability is preserved, the more natural it becomes for the user to edit the markup.

It should be noticed that TeXmacs also produces some auxiliary data as a byproduct of the typesetting product. For instance, the correct values of references and page numbers, as well as tables of contents, indexes, etc. are determined during the typesetting stage and memorized at a special place. Even though auxiliary data may be determined automatically from the document, it may be expensive to do so (one typically has to retypeset the document). When the auxiliary data are computed by an external plug-in, then it may even be impossible to perform the recomputations on certain systems. For these reasons, auxiliary data are carefully memorized and stored on disk when you save your work.

12.7. Data relation descriptions

The rationale behind D.R.D.s.

One major advantage of TeXmacs is that the editor uses general trees as its data format. Like for XML, this choice has the advantages of being simple to understand and making documents easy to manipulate by generic tools. However, when using the editor for a particular purpose, the data format usually needs to be restricted to a subset of the set of all possible trees.

In XML, one uses Data Type Definitions (D.T.D.s) in order to formally specify a subset of the generic XML format. Such a D.T.D. specifies when a given document is valid for a particular purpose. For instance, one has D.T.D.s for documents on the web (XHTML), for mathematics (MathML), for two-dimensional graphics (SVG) and so on. Moreover, up to a cetain extent, XML provides mechanisms for combining such D.T.D.s. Finally, a precise description of a D.T.D. usually also provides some kind of reference manual for documents of a certain type.

In TeXmacs, we have started to go one step further than D.T.D.s: besides being able to decide whether a given document is valid or not, it is also very useful to formally describe certain properties of the document. For instance, in an interactive editor, the numerator of a fraction may typically be edited by the user (we say that it is *accessible*), whereas the URL of a hyperlink is only editable on request. Similarly, certain primitives like itemize correspond to block content, whereas other primitives like sqrt correspond to inline content. Finally, certain groups of primitives, like chapter, section, subsection, etc. behave similarly under certain operations, like conversions.

A Data Relation Description (D.R.D.) consists of a Data Type Definition, together with additional logical properties of tags or document fragments. These logical properties are stated using so called *Horn clauses*, which are also used in logical programming languages such as Prolog. Contrary to logical programming languages, it should nevertheless be relatively straightforward to determine the properties of tags or document fragments, so that certain database techniques can be used for efficient implementations. At the moment, we only started to implement this technology (and we are still using lots of C++ hacks instead of what has been said above), so a more complete formal description of D.R.D.s will only be given at a later stage.

One major advantage of the use of D.R.D.s is that it is not necessary to establish rigid hierarchies of object classes like in object oriented programming. This is particularly useful in our context, since properties like accessability, inline-ness, etc. are quite independent one from another. In fact, where D.T.D.s may be good enough for the description of passive documents, more fine-grained properties are often usefull when manipulating documents in a more interactive way.

Current D.R.D. properties and applications.

Currently, the D.R.D. of a document contains the following information:

- The possible arities of a tag.
- The accessability of a tag and its children.

In the near future, the following properties will be added:

- Inline-ness of a tag and its children.
- Tabular-ness of a tag and its children.
- Purpose of a tag and its children.

The above information is used (among others) for the following applications:

- Natural default behaviour when creating/deleting tags or children (automatic insertion of missing arguments and removal of tags with too little children).
- Only traverse accessible nodes during searches, spell-checking, etc.
- Automatic insertion of document or table tags when creating block or tabular environments.
- Syntactic highlighting in source mode as a function of the purpose of tags and arguments.

Determination of the D.R.D. of a document.

TeXmacs associate a unique D.R.D. to each document. This D.R.D. is determined in two stages. First of all, TeXmacs tries to heuristically determine D.R.D. properties of user-defined tags, or tags which are defined in style files. For instance, when the user defines a tag like

```
⟨assign|hi|⟨macro|name|Hello name!⟩⟩
```

TeXmacs automatically notices that hi is a macro with one element, so it considers 1 to be the only possible arity of the hi tag. Notice that the heuristic determination of the D.R.D. is done interactively: when defining a macro inside your document, its properties will automatically be put into the D.R.D. (assuming that you give TeXmacs a small amount of free time of the order of a second; this minor delay is used to avoid compromising the reactivity of the editor).

Sometimes the heuristically defined properties are inadequate. For this case, TeXmacs provides the drd-props tag in order to manually override the default properties.

12.8. TeXmacs lengths

A simple TeXmacs length is a number followed by a length unit, like `1cm` or `1.5mm`. TeXmacs supports three main types of units:

Absolute units. The length of an absolute unit like `cm` or `pt` on print is fixed.

Context dependent units. Context-dependent length units depend on the current font or other environment variables. For instance, `1ex` corresponds to the height of the "x" character in the current font and `1par` correspond to the current paragraph width.

User defined units. Any nullary macro, whose name contains only lower case roman letters followed by `-length`, and which returns a length, can be used as a unit itself. For instance, the following macro defines the `dm` length:

```
⟨assign|dm-length|⟨macro|10cm⟩⟩
```

Furthermore, length units can be *stretchable*. A stretchable length is represented by a triple of rigid lengths: a minimal length, a default length and a maximal length. When justifying lines or pages, stretchable lengths are automatically sized so as to produce nicely looking layout.

In the case of page breaking, the *page-flexibility* environment provides additional control over the stretchability of white space. When setting the *page-flexibility* to 1, stretchable spaces behave as usual. When setting the *page-flexibility* to 0, stretchable spaces become rigid. For other values, the behaviour is linear.

Absolute length units.

`cm`. One centimeter.

`mm`. One millimeter.

`in`. One inch.

`pt`. The standard typographic point corresponds to 1/72.27 of an inch.

`bp`. A big point corresponds to 1/72 of an inch.

`dd`. The Didôt point equals 1/72 of a French inch, i.e. `0.376mm`.

`pc`. One "pica" equals 12 points.

`cc`. One "cicero" equals 12 Didôt points.

Rigid font-dependent length units.

`fs`. The font size. When using a `12pt` font, `1fs` corresponds to `12pt`.

`fbs`. The base font size. Typically, when selecting `10` as the font size for your document and when typing large text, the base font size is `10pt` and the font size `12pt`.

`ln`. The width of a nicely looking fraction bar for the current font.

`sep`. A typical separation between text and graphics for the current font, so as to keep the text readable. For instance, the numerator in a fraction is shifted up by `1sep`.

`yfrac`. The height of the fraction bar for the current font (approximately `0.5ex`).

`ex`. The height of the "x" character in the current font.

`emunit`. The width of the "M" character in the current font.

Stretchable font-dependent length units.

`fn`. This is a stretchable variant of `1quad`. The default length of `1fn` is `1quad`. When stretched, `1fn` may be reduced to `0.5fn` and extended to `1.5fn`.

`fns`. This length defaults to zero, but it may be stretched up till `1fn`.

`bls`. The "base line skip" is the sum of `1quad` and *par-sep*. It corresponds to the distance between successive lines of normal text.

Typically, the baselines of successive lines are separated by a distance of `1fn` (in TeXmacs and LaTeX a slightly larger space is used though so as to allow for subscripts and superscripts and avoid a too densely looking text. When stretched, `1fn` may be reduced to `0.5fn` and extended to `1.5fn`.

`spc`. The (stretchable) width of space character in the current font.

`xspc`. The additional (stretchable) width of a space character after a period.

Other length units.

`par`. The width of the paragraph. That is the length the text can span. It is affected by paper size, margins, number of columns, column separation, cell width (if in a table), etc.

`pag`. The height of the main text in a page. In a similar way as `par`, this length unit is affected by page size, margins, etc.

`px`. One screen pixel, the meaning of this unit is affected by the shrinking factor.

`tmpt`. The smallest length unit for internal length calculations by TeXmacs. `1px` divided by the shrinking factor corresponds to `256tmpt`.

Different ways to specify lengths.

There are three types of lengths in TeXmacs:

Simple lengths. A string consisting of a number followed by a length unit.

Abstract lengths. An abstract length is a macro which evaluates to a length. Such lengths have the advantage that they may depend on the context.

Normalized lengths. All lengths are ultimately converted into a normalized length, which is a tag of the form ⟨tmlen|*l*⟩ (for rigid lengths) or ⟨tmlen|*min*|*def*|*max*⟩ (for stretchable lengths). The user may also use this tag in order to specify stretchable lengths. For instance, ⟨tmlen|⟨minus|1quad|1pt⟩|1quad|1.5quad⟩ evaluates to a length which is `1quad` by default, at least `1quad-1pt` and at most `1.5quad`.

Chapter 13

Built-in environment variables

The way $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ typesets documents is influenced by so called *environment variables*. The style-sheet language uses a so called *environment* (or context) to store both environment variables and macros. The environment variables are subdivided into two catagories: built-in variables and additional variables provided by style files. Built-in variables usually affect the layout, while additional variables mostly serve computational purposes. In the next sections of this chapter, we will describe all built-in environment variables.

A typical built-in environment variable is *color*. The value of an environment variable may be changed permanently using assign and temporarily using the with primitive:

> Some colored text.

> Some ⟨with|*color*|dark red|colored⟩ text.

Counters are typical environment variables defined in style-sheets.

> 1. A weirdly
>
> 4. numbered list...

> ⟨enumerate|
>   ⟨item⟩A weirdly
>   ⟨assign|*item-nr*|3⟩⟨item⟩numbered list...⟩

The typesetting language uses *dynamic scoping* of variables. That means that macros can access and modify variables in their calling context. In the previous example, the enumerate macro locally initializes *item-nr* to 0 (uses with) and the item macro increments it by one and shows its value. Since enumerate locally redefines *item-nr*, the original value of *item-nr* is restored on exit.

Each document comes with an initial environment with the initial values of environment values, i.e. their values just before we typeset the document. If an environment variable does not occur in the initial environment, then its initial value defaults to its value after typesetting the document style and possible additional packages. The initial environment before typesetting the style files and packages is built-in into the editor.

Some variables, like header and footer variables, must be set inside the document, their initial environment value is ignored. Generally, they should be set by header and sectioning markup.

13.1. General environment variables

mode := `text` (major mode)

This very important environment variable determines the *current mode*. There are four possible values: `text` (text mode), `math` (mathematical mode), `prog` (programming mode) and `src` (source mode). The behaviour of the editor (menus, keystrokes, typesetting, etc.) depends heavily on the mode. For example, the following code may be used in order to include a mathematical formula inside text:

The formula $a^2 + b^2 = c^2$ is well known.

The formula ⟨with|*mode*|math|a⟨rsup|2⟩+b⟨rsup|2⟩=c⟨rsup|2⟩⟩ is well known.

Some other environment variables (mainly the language and the font) also depend on the current mode (in this context, the source mode always behaves in a similar way as the text mode). During copy&paste and search&replace operations, TeXmacs tries to preserve the mode.

language := `english`
math-language := `texmath`
prog-language := `scheme` (language)

A second major environment variable is the *current language*. In fact, there are three such environment variables: one for each mode. The language in which content is written is responsible for associating a precise semantics to the content. This semantics is used for different purposes:

- The language may specify rules for typesetting content. For instance, the text language specifies punctuation and hyphenation rules. Similarly the mathematical language containns spacing information for mathematical operators.

- Several editing operations depend on the current language: when performing a search or replace operation, TeXmacs is both mode and language sensitive. Similarly, the text language determines the dictionary to use when spell-checking the document.

- The language controls (among other parameters like the mode and the document format) the way content is being converted from one context to another.

  Currently, no real language-dependent conversions have been implemented yet. But in the future one may imagine that copying a piece of English text to a document written in French will perform an automatic translation. Similarly, a mathematical document might be converted from infix to postfix notation.

- The programming language determines the current scripting language in use. Other scripting languages than Scheme are currently only used for interactive sessions, but primitives like `extern` and `mutator` might become language-sensitive in the future.

At the moment, the current language is mainly used as a hint for indicating the semantics of text: it is not required that a text written in English contains no spelling errors, or that a formula written in a mathematical language is mathematically or even syntactically correct. Nevertheless, the editor is intended to enforce correctness more and more, especially for mathematics.

The language may be specified globally for the whole document in Document → Language and locally for a piece of text in Format → Language.

prog-session := `default` (name of programming session)

This environment variables is used in addition to the *prog-language* variable in order to determine a concrete implementation as well as a particular instance of the current programming language. For instance, in case of the MAXIMA language, different implementation may be used fooor the underlying LISP. Similarly, one may wish to run two different instances of MAXIMA in parallel.

magnification := `1` (magnification)

This variable determines the magnification which is applied to all content. Magnifications bigger than one are typically useful for presentations (from slides or from a laptop):

normal big huge

normal⟨htab | 5mm⟩⟨with | *magnification* | 2 | big⟩⟨htab | 5mm⟩⟨with | *magnification*|3|huge⟩

The magnification should not be confused with the font size: contrary to the magnification, the font size may also affect the shapes of the glyphs. The magnification is usually specified for the entire document in Document → Magnification.

bg-color := `white` (background color)

The background color for your document, as specified in Document → Color → Background.

color := `black` (foreground color)

The current foreground color of text and graphics, as specified in Document → Color → Foreground or Format → Color.

preamble := `false` (edit source tree?)

This flag determines whether we are editing normal text or a style-sheet. The source tree or preamble mode may be selected in Document → View → Edit source tree.

info-flag := `short` (informative flags style)

This variable controls the rendering of informative flags, which are for instance used to indicate the locations of otherwise invisible labels or typesetting directives. The *info-flag* may take the values `none`, `short` and `detailed`:

Label 1, Label 2, Label 3.

⟨with|*info-flag*|none|Label 1⟨label|flag-label-1⟩⟩, ⟨with|*info-flag*|short|Label 2⟨label|flag-label-2⟩⟩, ⟨with|*info-flag*|detailed|Label 3⟨label|flag-label-3⟩⟩.

Usually, the rendering of informative flags is specified document-wide in Document → View → Informative flags.

13.2. Specifying the current font

In this section, we describe the environment variables which control the rendering of fonts. Several parameters may be defined independently for each mode (the font name, variant, series and shape), whereas other parameters are uniform for all modes. Font properties may be controlled globally for the whole document in Document → Font and locally for document fragments in Format → Font.

From an abstract point of view, a *font* is defined to be a graphically consistent way of rendering strings. Fonts are usually made up from glyphs like "x", "ffi", "α", "$\sum$", etc. When rendering a string, the string is decomposed into glyphs so as to take into account ligatures (like fi, fl, ff, ffi, ffl). Next, the individual glyphs are positioned while taking into account kerning information (in "xo" the "o" character is slightly shifted to the left so as to take profit out of the hole in the "x"). In the case of mathematical fonts, TEXMACS also provides a coherent rendering for resizable characters, like the large brackets in

$$\Big(\Big(\Big(\Big)\Big)\Big).$$

Similarly, a *font family* is a family of fonts with different characteristics (like font weight, slant, etc.), but with a globally consistent rendering. One also says that the fonts in a font family "mix well together". For instance, the standard computer modern roman font and its **bold** and *italic* variants mix well together, but the computer modern roman font and the Avant Garde font do not.

REMARK 13.1. For the future, it is planned to replace the font variant and font shape variables by a larger range of properties to individually control the slant, serifs, small-caps, and so on. It is also planned to systematically use Unicode fonts with possible additional glyphs for mathematics. This should automatically enable the use of Cyrillic characters inside Russian text and similarly for other languages.

font := `roman`
math-font := `roman`
prog-font := `roman` (font name)

These variables control the main name of the font, also called the *font family*. For instance:

Computer modern roman, Pandora, *Chancery*, Palatino

Similarly, TEXMACS supports various mathematical fonts:

Roman: $a^2 + b^2 = c^2$

Adobe: $a^2 + b^2 = c^2$

New roman: $a^2 + b^2 = c^2$

Concrete: $a^2 + b^2 = c^2$

font-family := `rm`
math-font-family := `mr`
prog-font-family := `tt` (font variant)

This variable selects a variant of the major font, like a sans serif font, a typewriter font, and so on. As explained above, variants of a given font are designed to mix well together. Physically speaking, many fonts do not come with all possible variants (sans serif, typewriter, etc.), in which case TeXmacs tries to fall back on a suitable alternative font.

Typical variants for text fonts are `rm` (roman), `tt` (typewriter) and `ss` (sans serif):

roman, `typewriter` and sans serif

In maths mode, a distinction is made between the mathematical variants `mr` (roman), `mt` (typewriter) and `ms` (sans serif) and textual variants `rm` (roman), `bf` (bold), etc. In the first case, variables and operators are usually rendered in a different slant, contrary to the second case:

ms: $\sin (x + y) = \sin x \cos y + \cos x \sin y$

ss: $\sin (\mathrm{x} + \mathrm{y}) = \sin \mathrm{x} \cos \mathrm{y} + \cos \mathrm{x} \sin \mathrm{y}$

font-series := `medium`
math-font-series := `medium`
prog-font-series := `medium` (font weight)

The font series determines the weight of the font. Most fonts only provide `regular` and `bold` font weights. Some fonts also provide `light` as a possible value.

medium, **bold**

font-shape := `right`
math-font-shape := `normal`
prog-font-shape := `right` (font shape)

The font shape determines other characters of a font, like its slant, whether we use small capitals, whether it is condensed, and so on. For instance,

upright, *slanted*, *italic*, left slanted, Small Capitals, proportional typewriter, **bold condensed**, flat sans serif, long

font-base-size := `10` (font base size)

The base font size is specified in `pt` units and is usually invariant throughout the document. Usually, the base font size is `9pt`, `10pt`, `11pt` or `12pt`. Other font sizes are usually obtained by changing the *magnification* or the relative font-size.

9pt, 10pt, 11pt, 12pt

font-size := 1 (font size)

The real font size is obtained by multiplying the *font-base-size* by the *font-size* multiplier. The following standard font sizes are available from Format → Size:

size	multiplier	size	multiplier
Tiny	0.59	Very small	0.71
Small	0.84	Normal	1
Large	1.19	Very large	1.41
Huge	1.68	Really huge	2

Table 13.1. Standard font sizes.

From a mathematical point of view, the multipliers are in a geometric progression with factor $\sqrt[4]{2}$. Notice that the font size is also affected by the index level.

dpi := 600 (fonts rendering quality)

The rendering quality of raster fonts (also called Type 3 fonts), such as the fonts generated by the METAFONT program is controlled through its discretization precision in dots per inch. Nowadays, most laser printers offer a printing quality of at least `600dpi`, which is also the default *dpi* setting for TeXmacs. For really high quality printing, professionals usually use a precision of `1200dpi`. The *dpi* is usually set once and for all for the whole document.

13.3. Typesetting mathematics

math-level := 0 (index level)

The *index level* increases inside certain mathematical constructs such as indices and fractions. When the index level is high, formulas are rendered in a smaller font. Nevertheless, index levels higher than 2 are all rendered in the same way as index level 2; this ensures that formulas like

$$\mathrm{e}^{\mathrm{e}^{\mathrm{e}^{\mathrm{e}^{x}}}} = \frac{1+\frac{1}{x+\mathrm{e}^{x}}}{1+\frac{1}{\mathrm{e}^{x}+\frac{1}{\mathrm{e}^{\mathrm{e}^{x}}}}}$$

remain readable. The index level may be manually changed in Format → Index level, so as to produce formulas like

$$x^{y^{z}}$$

x⟨rsup|⟨with|*math-level*|0|y⟨rsup|⟨with|*math-level*|0|z⟩⟩⟩⟩

math-display := `false` (display style)

This environement variable controls whether we are in *display style* or not. Formulas which occur on separate lines like

$$\frac{n}{H(\alpha_1,\ldots,\alpha_n)}=\frac{1}{\alpha_1}+\cdots+\frac{1}{\alpha_n}$$

are usually typeset in display style, contrary to inline formulas like $\frac{n}{H(\alpha_1,\ldots,\alpha_n)}=\frac{1}{\alpha_1}+\cdots+\frac{1}{\alpha_n}$. As you notice, formulas in display style are rendered using a wider spacing. The display style is disabled in several mathematical constructs such as scripts, fractions, binomial coefficients, and so on. As a result, the double numerators in the formula

$$H(\alpha_1,\ldots,\alpha_n)=\frac{n}{\frac{1}{\alpha_1}+\cdots+\frac{1}{\alpha_n}}$$

are typeset in a smaller font. You may override the default settings using Format → Display style.

math-condensed := `false` (condensed display style)

By default, formulas like $a+\cdots+z$ are typeset using a nice, wide spacing around the $+$ symbol. In formulas with scripts like $\mathrm{e}^{a+\cdots+z}+\mathrm{e}^{\alpha+\cdots+\zeta}$ the readability is further enhanced by using a more condensed spacing inside the scripts: this helps the reader to distinguish symbols occurring in the scripts from symbols occurring at the ground level when the scripts are long. The default behaviour can be overridden using Format → Condensed.

math-vpos := 0 (position in fractions)

For a high quality typesetting of fraction, it is good to avoid subscripts in numerators to descend to low and superscripts in denominators to ascend to high. TeXmacs therefore provides an additional environment variable *math-vpos* which takes the value 1 inside numerators, -1 inside denominators and 0 otherwise. In order to see the effect the different settings, consider the following formula:

$$a_{-1}^2+a_0^2+a_1^2$$

⟨with|*math-vpos*|-1|⟨group|a_{-1}^2⟩⟩+⟨with|*math-vpos*|0|⟨group|a_0^2⟩⟩+⟨with|*math-vpos*|1|⟨group|a_1^2⟩⟩

In this example, the grouping is necessary in order to let the different vertical positions take effect on each a_i^2. Indeed, the vertical position is uniform for each horizontal concatenation.

13.4. Paragraph layout

par-mode := `justify` (paragraph alignment)

This environment variable specifies the alignment of the different lines in a paragraph. Possible values are `left`, `center`, `right` and `justify`:

This paragraph is aligned to the left. This paragraph is aligned to the left. This paragraph is aligned to the left.	This paragraph is has been centered. This paragraph is has been centered. This paragraph is has been centered.
This paragraph is aligned to the right. This paragraph is aligned to the right. This paragraph is aligned to the right.	This paragraph has been justified. Justification is the default alignment mode for paragraphs. So be it.

Table 13.2. The supported modes for alignment.

par-hyphen := `normal` (quality of hyphenation)

This parameter controls the quality of the hyphenation algorithm. Possible values are `normal` and `professional`. The professional hyphenation algorithm uses a global algorithm on the entire paragraph, whereas the normal one uses a faster first-fit algorithm.

The difference between the different hyphenation algorithms provided by $\mathrm{T_EX_{MACS}}$ is seen best for long paragraphs which are typeset into a narrow column. The professional hyphenation usually succeeds to minimize the number of ugly gaps between words.	The difference between the different hyphenation algorithms provided by $\mathrm{T_EX_{MACS}}$ is seen best for long paragraphs which are typeset into a narrow column. The professional hyphenation usually succeeds to minimize the number of ugly gaps between words.

Table 13.3. Comparison different hyphenation algorithms. At the left hand side, we have used the normal algorithm and on the right hand side the professional one. Even though there are some ugly gaps at the right hand side around "hyphenation", the really bad gap around "The" on the left hand side has been avoided.

par-width := `auto` (paragraph with)

This environment variable controls the width of paragraphs. By default, it is automatically determined as a function of the page (or screen) size and margins.

par-left := `0cm`
par-right := `0cm` (left and right margins)

These environment variables specify absolute left and right margins for the paragraph, with respect to the default left and right margins (which are determined as a function of the page layout). For instance:

This text uses the default margins.

> This text uses a left margin of `1cm`
>
> > This text uses a left margin of `2cm`
> >
> > > This text uses a left margin of `3cm`
> > >
> > > The left and right margins of this text have both been set to `3cm`.

Environments like itemize and quote-env which maybe nested usually compute new margins as a function of the old values by adding or subtracting some space:

```
⟨assign|quote-env|
  ⟨macro|body|
    ⟨surround|
      ⟨vspace*|0.5fn⟩|
      ⟨right-flush⟩⟨vspace|0.5fn⟩|
      ⟨with|par-left|⟨plus|par-left|3fn⟩|par-right|⟨plus|par-right|
      3fn⟩|par-first|0fn|par-par-sep|0.25fn|body⟩⟩⟩⟩
```

par-first := `1.5fn` (first indentation)

The *par-first* parameter specifies the additional indentation which is used for the first line of the paragraph. The aim of first indentations is to indicate the starts of new paragraphs. An alternative technique is the use of vertical whitespace.

The `article` and `book` styles in TeXmacs indictate the starts of new paragraphs through the use of a first indentation. The `generic` and `letter` styles rather use vertical whitespace.	The `generic` and `letter` styles in TeXmacs indictate the starts of new paragraphs through the use of vertical whitespace. The `article` and `book` styles rather use a first indentation.

Table 13.4. Two classical ways to indicate the starts of new paragraphs.

par-sep := `0.2fn` (extra separation between successive lines)

The sum of the font size and *par-sep* determines the ideal distance between two successive base lines in a paragraph (also called the "base line skip"). Of course, when the lines contain large boxes, then this distance may need to be increased. When `1fn` for *par-sep*, one may for instance produce documents with a double interline space:

> A double interline space corresponds to *par-sep* := `1fn`. Double interline spaces are often used by lazy people who want to pretend that they have written many pages. They generally do not care about tropical rain forests.

In the case when two successive lines use different base line skips, then the maximal value is used in order to compute the ideal distance between their baselines. This allows for a reasonable spacing when the font size is changed from one paragraph to another:

> Normal text.
>
> ## Some very large text.
>
> And back to normal.

par-line-sep := `0.025fn*` (extra space between lines)

This parameter corresponds an additional stretchable amount of whitespace between successive lines in a paragraph. Setting *par-line-sep* to a small stretchable value which defaults to 0 allows the page breaker to correctly stretch pages which contain a very long textual paragraph. Indeed, *par-line-sep* vanishes, then the height of a textual paragraph is of the form $a + b\,n$, where a and b are constants and n is the number of lines. There is no reason why the usable height of a page should be of this form.

par-par-sep := `0.5fn*` (extra space between paragraphs)

The *par-par-sep* parameter specifies the amount of vertical whitespace which separates two successive paragraphs. This space is determined in stretchable length units. By default, $\mathrm{T_EX_{MACS}}$ does not use any whitespace between successive paragraphs, except when no nice page breaks could be found (this explains the use of the `fn*` length unit). Starts of new paragraphs are rather indicated through the use of first indentations (see table 13.4).

In the case when two successive paragraph use different paragraph separations, then the maximum of the two is taken. In fact, the *par-par-sep* length is added to both the vertical spacing before and the vertical spacing after the paragraph.

par-hor-sep := `0.5fn`
par-ver-sep := `0.2fn` (minimal space between ink)

When a paragraph contains several exceptionally large boxes, then $\mathrm{T_EX_{MACS}}$ attempts to "shove successive lines into another" as long as none of the boxes collide:

> Consider a fraction which decends more than usual like $\frac{1}{x+1}$ at the end of a line and an expression like $\mathrm{e}^{\mathrm{e}^x}$ which is higher than usual.
>
> When these expressions occur at different places, then $\mathrm{T_EX_{MACS}}$ tries to render the successive lines in a compact manner.
>
> In the case of a fraction $\frac{1}{x+1}$ and an exceptionally high expression at the wrong place, like the expression $\mathrm{e}^{\mathrm{e}^x}$ here, the boxes are separated by *env-ver-sep*.

As soon as the horizontal distance between two large boxes is less than *par-hor-sep*, then they are considered to be in collision. In that case, the vertical distance between them must be at least *par-ver-sep*. Also, the amount of showing never exceeds `1ex`.

When using an interline space of `1.5` or `2`, the default value of *par-ver-sep* allows the user to type larger formulas in the text while preserving a uniform layout. When using a small *par-sep* and a large *par-ver-sep*, the distance between two successive lines remains small, except when their contents are horizontally close. This may for instance be used to reduce the space between a short like followed by a centered equation.

par-fnote-sep := `0.2fn` (minimal space between different footnotes)

This parameter controls the amount of vertical space between successive footnotes.

par-columns := `1` (number of columns)

This environment variable specifies the number of columns into which the text is being typeset. Different numbers of columns may be used successively in the same document.

par-columns-sep := `2fn` (distance between columns)

This environment variable specifies the amount of horizontal whitespace which separates different columns in multi-column mode.

13.5. PAGE LAYOUT

In this section, we describe how $\mathrm{T_EX_{MACS}}$ fills pages with typesetted content. Besides specifying the settings on how to print a document, the user may also determine the way pages should be rendered on screen. It should be noticed that the number of environment variables is redundant in the sense that some variables are computed as a function of other ones. For instance, by default, the paragraph width is computed as a function of the page size and the left and right margins.

Paper specific variables.

page-type := `a4` (the size of pages)

Specify the size of a page when printing out. Most standard formats are available in Document → Page → Size. By default, the paper size is the one of your printer (the default printer settings may be changed in Edit → Preferences → Printer). When the *page-type* is set to `user`, then the page size is given by *page-width* and *page-height*.

page-orientation := `portrait` (page orientation)

The orientation of pages can be either `portrait` or `landscape`.

page-nr := `0` (current page number)

The current page number. This environment variable should be manipulated with care, since it is not yet available at typesetting time. For a reliable determination of page numbers, one may combine the label and page-ref primitives. Nevertheless, the *page-nr* variable can be used in the macros which render page headers and footers.

page-the-page (display the page number)

This environment variable really contains the macro which is used for rendering the page-number. By default, it renders *page-nr*. The macro takes no arguments. In order to simulate a document whose first page number os 123, one may redefine

〈assign|*page-the-page*|〈macro|〈plus|*page-nr*|122〉〉〉

page-breaking := `optimal` (page breaking algorithm)

This parameter specifies the page breaking algorithm. The default `optimal` algorithm takes into account the global document and tries hard to avoid bad page breaks. The alternative `sloppy` algorithm uses a fast first-fit algorithm, but produces bad page break with a higher probability. The `medium` quality algorithm is the same as the `optimal` algorithm, except for two column content.

page-flexibility := `1.0` (flexibility for stretching)

This parameter specifies how much stretchable spaces may be extended or reduced in order to fill pages which are too short or too long. A page flexibility of 1 allows spaces to be stretched to their minimal and maximal values. A page flexibility of 0 prevents spaces to be stretched. For other values of *page-flexibility* the behaviour is linear.

page-shrink := `1fn` (allowed amount of page shrinking)

In the case when it is very hard to find good page breaks, this parameter specifies an additional amount of space by which a page is allowed to be reduced.

page-extend := `0fn` (allowed amount of page extensions)

In the case when it is very hard to find good page breaks, this parameter specifies an additional amount of space by which a page is allowed to be extended.

Screen specific variables.

page-medium := `papyrus` (the page medium)

This environment variable, which is initialized using Document → Page → Type, specifies how pages are rendered on the screen. The following values are available:

`paper`. Page breaks are visually indicated on the screen. This mode is useful for ajusting the final version of a document before printing or sending it to a publisher. However, the use of this mode slows down the editor since every modification in the document triggers the page-breaking algorithm.

Notice also that the mere selection of this mode does not imply the screen margins and page decorations to be as on paper. In order to previsualize a document in a fully realistic way, you should also set Document → View → Page layout → Show header and footer and Document → View → Page layout → Margins as on paper.

`papyrus`. The paragraph width is the same as on paper, but page breaking is disabled. This mode is most useful during the editing phase of a document which will ultimately be printed out. It combines a reasonable editing speed with realistic line breaks.

`automatic`. The paragraph width is as large as possible so as to fit into the current window and page breaking is disabled. This setting, which makes optimal use of the available space on your screen, is useful for documents which are not intended to be printed out. It may for instance be selected when using TeXmacs as a browser or as an interface to computer algebra systems.

page-screen-width := `10cm` (width of the rendering window)

In `automatic` mode, this environement variable contains the width of the screen.

page-screen-height := `10cm` (height of the rendering window)

In `automatic` mode, this environement variable contains the height of the screen.

page-screen-margin := `true` (special margins for screen editing?)

This flag specifies whether the screen margins are manually specified by the user, or whether they are the same as on paper.

page-screen-left := `5mm`
page-screen-right := `5mm`
page-screen-top := `15mm`

page-screen-bot := `15mm` (left margin on screen)

When *page-screen-margin* is `true`, then these environment variables determine the margins which are to be used for rendering on the screen.

page-show-hf := `false` (show headers and footers on screen?)

This flag determines whether the page headers and footers should be visible on the screen. When set to `true`, it should be noticed that the headers and footers are not always correctly updated when editing. In the case when you suspect them to be wrong, refreshing the display by scrolling down and up should display the correct values.

Specifying the margins.

The parameters for page margins are represented schematically at the left hand side in figure 13.1. One may either specify the paragraph width as a function of the left and right margins, or *vice versa*. The left and right margins may depend on whether the page number is odd or even.

page-width-margin := `false`
page-height-margin := `false` (compute margins from main text dimensions?)

When *page-width-margin* is set to `false`, then the paragraph width *par-width* is determined automatically from the page size and the left and right margins. When set to `true`, the left and right margins are determined as a function of the page size, the paragraph width, *page-odd-shift* and *page-even-shift*. For compatability with TEX/LATEX, it is also possible to set *page-width-margin* to `tex`, in which case the horizontal margins are determined from *page-odd*, *page-even* and *par-width*. The *page-height-margin* variable plays a similar role for the vertical margins.

page-width := `auto`
page-height := `auto` (page width)

By default, the width and height of a page are automatically determined from the page type. When *page-type* is set to `user`, then the user may manually specify the page size using *page-width* and *page-height*.

page-odd := `auto`
page-even := `auto` (left margin)

If *page-width-margin* is set to `false`, then *page-odd* and *page-even* specify the left margins for odd and even pages. If *page-width-margin* is `true`, then these values are computed as a function of the page size, the paragraph width, *page-odd-shift* and *page-even-shift*. When *page-odd* and *page-even* are set to `auto`, then a nice default left margin is determined as a function of the specified page type.

page-right := `auto` (right margin)

If *page-width-margin* is set to `false`, then *page-right* specifies the right margin for odd pages. The right margin for even pages is given by the formula

$$\textit{page-right} + \textit{page-even} - \textit{page-odd}$$

If *page-width-margin* is `true` or when *page-right* is set to `auto`, then the right margin is determined in a similar way as the left margin.

page-odd-shift := `0mm`

page-even-shift := `0mm` (margin shifts)

If *page-width-margin* is set to `true`, then the left margins for odd and even pages are determined from the page size, paragraph width and the margin shifts using the formulas

$$\begin{aligned} \textit{page-even} &= \frac{\textit{page-width} - \textit{par-width}}{2} + \textit{page-odd-shift} \\ \textit{page-odd} &= \frac{\textit{page-width} - \textit{par-width}}{2} + \textit{page-even-shift} \end{aligned}$$

The right margin is always taken to be such that the paragraph width and the left and right margins sum up to the page width.

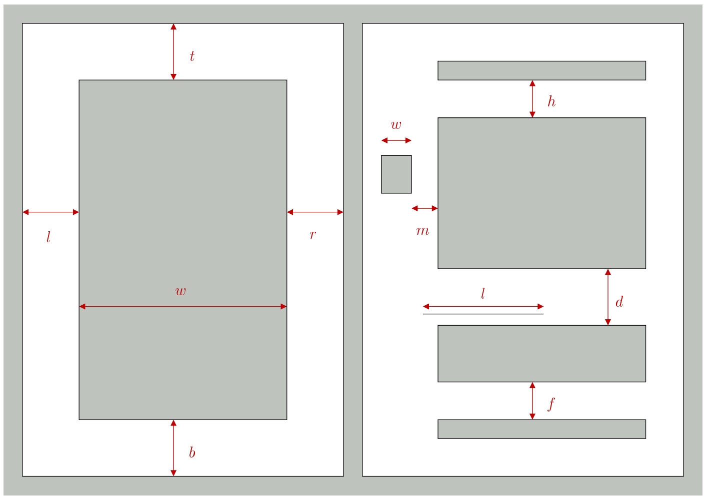


Figure 13.1. Schematic representation of the layout of pages. On the left hand side, the parameters l, r, t and b respectively correspond to the left, right, top and bottom margins, and w corresponds to the paragraph width. On the right hand side, h, f, d and m correspond to the header, footer, footnote and marginal note separations, w to the width of marginal notes, and l to the length of the footnote bar.

Page decorations.

page-odd-header :=
page-odd-footer :=
page-even-header :=
page-even-footer := (header for odd pages)

These environment variables contain the header and footer texts for odd and even pages.

page-head-sep := `8mm`
page-foot-set := `8mm` (separation between headers/fotters and text)

These parameters determine the space between the main text and page headers and footers. They correspond to the h and f distances at the right hand side of figure 13.1.

page-fnote-sep := `1.0fn` (space between footnotes and text)

The separation between the main text and footnotes, i.e. the distance d in figure 13.1.

page-fnote-barlen := `7.5fn` (length of footnote bars)

The length of the foornote bar.

page-float-sep := `1.5fn` (separation between floats and text)

The separation between the main text and floating objects.

page-mnote-sep := `5mm` (separation between marginal notes and text)

The separation between marginal notes and the main text (not implemented yet).

page-mnote-width := `15mm` (width of marginal notes)

The width of marginal notes (not implemented yet).

13.6. Table layout

The environment variables for tables can be subdivided in variables (prefixed by *table-*) which apply to the whole table and those (prefixed by *cell-*) which apply to individual cells. Whereas usual environment variables are set with assign and with, the tabular environment variables are rather set with the tformat primitive. This makes it possible to apply certain settings to any rectangular subtable of the entire table and in particular to rows or columns. For more details, see the documentation of the twith and cwith primitives.

Layout of the table as a whole.

table-width :=
table-height := (hint for table dimensions)

These parameters indicate a hint for the dimensions of the table. The *table-hmode* and *table-vmode* variables determine how to take into account these settings.

table-hmode :=
table-vmode := (determination of table dimensions)

These parameters specify how to determine the dimensions of the table. At the moment, the values of *table-hmode* and *table-vmode* are actually ignored and *table-width* and *table-height* are interpreted as the minimal width and height of the table.

table-halign := `l`
table-valign := `f` (alignment inside text)

These parameters determine how the table should be aligned in the surrounding text. Possible values for *table-halign* are `l` (left), `c` (center) and `r` (right), and possible values for *table-valign* are `t` (top), `f` (centered at fraction bar height), `c` (center) and `b` (bottom).

In addition to the above values, the alignment can take place with respect to the baselines of particular cells. Such values for *table-halign* are `L` (align w.r.t. the left column), `C` (align w.r.t. the middle column), `R` (align w.r.t. the right column) and `O` (align w.r.t. the priviledged origin column *table-col-origin*). Similarly, *table-halign* may take the additional values `T` (align w.r.t. the top row), `C` (align w.r.t. the middle row), `B` (align w.r.t. the bottom row) and `O` (align w.r.t. the priviledged origin row *table-row-origin*).

table-row-origin := `0`
table-col-origin := `0` (priviledged cell)

Table coordinates of an priviledged “origin cell” which may be used for aligning the table in the surrounding text (see above).

table-lsep := `0fn`
table-rsep := `0fn`
table-bsep := `0fn`
table-tsep := `0fn` (padding around table)

Padding around the table (in addition to the padding of individual cells).

table-lborder := `0ln`
table-rborder := `0ln`
table-bborder := `0ln`
table-tborder := `0ln` (border around table)

Border width for the table (in addition to borders of the individual cells).

table-hyphen := `n` (allow for hyphenation?)

A flag which specifies whether page breaks may occur at the middle of rows in the table. When *table-hyphen* is set to `y`, then such page breaks may only occur when

1. The table is not surrounded by other markup in the same paragraph.
2. The rows whether the page break occurs has no borders.

An example of a tabular environment which allows for page breaks is `eqnarray*`.

table-min-rows :=
table-min-cols :=
table-max-rows :=
table-max-cols := (constraints on the table's size)

It is possible to specify a minimal and maximal numbers of rows or columns for the table. Such settings constraint the behaviour of the editor for operations which may modify the size of the table (like the insertion and deletion of rows and columns). This is particularly useful for tabular macros. For instance, *table-min-columns* and *table-max-columns* are both set to 3 for the `eqnarray*` environment.

Layout of the individual cells.

cell-background := (background color)

A background color for the cell.

cell-width :=
cell-height := (hint for cell dimensions)

Hints for the width and the height of the cell. The real width and height also depend on the modes *cell-hmode* and *cell-vmode*, possible filling (see *cell-hpart* and *cell-vpart* below), and, of course, on the dimensions of other cells in the same row or column.

cell-hpart :=
cell-vpart := (fill part of unused space)

When the sum s of the widths of all columns in a table is smaller than the width w of the table itself, then it should be specified what should be done with the unused space. The *cell-hpart* parameter specifies a part in the unsused space which will be taken by a particular cell. The horizontal part taken by a column is the maximum of the horizontal parts of its composing cells. Now let p_i the so determined part for each column ($i \in \{1, \ldots, n\}$). Then the extra horizontal space which will be distributed to this column is $p_i (w - s)/(p_1 + \cdots + p_n)$. A similar computation determines the extra vertical space which is distributed to each row.

cell-hmode := `exact`
cell-vmode := `exact` (determination of cell dimensions)

These parameters specify how to determine the width and the height of the cell. If *cell-hmode* is `exact`, then the width is given by *cell-width*. If *cell-hmode* is `min` or `max`, then the width is the minimul resp. maximum of *cell-width* and the width of the content. The height is determined similarly.

cell-halign := `l`
cell-valign := `B` (cell alignment)

These parameters determine the horizontal and vertical alignment of the cell. Possible values of *cell-halign* are `l` (left), `c` (center), `r` (right), `.` (decimal dot), `,` (decimal comma) and `R` (vertical baseline). Possible values of *cell-valign* are `t` (top), `c` (center), `b` (bottom) and `B` (baseline).

cell-lsep := `0fn`
cell-rsep := `0fn`
cell-bsep := `0fn`
cell-tsep := `0fn` (cell padding)

The amount of padding around the cell (at the left, right, bottom and top).

cell-lborder := `0ln`
cell-rborder := `0ln`
cell-bborder := `0ln`
cell-tborder := `0ln` (cell borders)

The borders of the cell (at the left, right, bottom and top). The displayed border between cells $T_{i,j}$ and $T_{i,j+1}$ at positions (i,j) and $(i,j+1)$ is the maximum of the borders between the right border of $T_{i,j}$ and the left border of $T_{i,j+1}$. Similarly, the displayed border between cells $T_{i,j}$ and $T_{i+1,j}$ is the maximum of the bottom border of $T_{i,j}$ and the top border of $T_{i+1,j}$.

cell-vcorrect := `a` (vertical correction of text)

As described above, the dimensions and the alignment of a cell may depend on the dimensions of its content. When cells contain text boxes, the vertical bounding boxes of such text may vary as a function of the text (the letter “k” resp. “y” ascends resp. descends further than “x”). Such differences sometimes leads to unwanted, non-uniform results. The vertical cell correction allows for a more uniform treatment of text of the same font, by descending and/or ascending the bounding boxes to a level which only depends on the font. Possible values for *cell-vcorrect* are `n` (no vertical correction), `b` (vertical correction of the bottom), `t` (vertical correction of the top), `a` (vertical correction of bottom and the top).

cell-hyphen := `n` (allow for hyphenation inside cells)

By default, the cells contain inline content which is not hyphenated. By selecting Table → Special cell properties → Hyphenation → Multi-paragraph, the cell contents becomes multi-paragraph. In that case, *cell-hyphen* determines how this content is hyphenated. Possible values are `n` (disable line breaking) and `b`, `c` and `t` (enable line breaking and align at the bottom, center resp. top line).

cell-row-span := `1`

cell-col-span := 1 (span of a cell)

Certain cells in a table are allowed to span over other cells at their right or below them. The *cell-row-span* and *cell-col-span* specify the row span and column span of the cell.

cell-decoration := (decorating table for cell)

This environment variable may contain a decorating table for the cell. Such a decoration enlarges the table with extra columns and cells. The tmarker primitive determines the location of the original decorated cell and its surroundings in the enlarged table are filled up with the decorations. Cell decorations are not really used at present and may disappear in future versions of TeXmacs.

cell-orientation := `portrait` (orientation of cell)

Other orientations for cells than `portrait` have not yet been implemented.

cell-row-nr := 1
cell-col-nr := 1 (current cell position)

In the future, these environment variables should contain the current cell position during the typesetting process.

13.7. Editing source trees

The different rendering styles for source trees are described in more detail in the section about the global presentation of source trees. The corresponding environment variables are briefly described here.

src-style := `angular` (rendering style for source tags)

The principal rendering style for source trees as specified in Document → View → Style. Possible values are `angular`, `scheme`, `functional` and `latex`.

src-special := `normal` (how to render special tags)

How to render special tags like concat, document, compound, etc., as specified in Document → View → Special. Possible values are `raw`, `format`, `normal` and `maximal`.

src-compact := `normal` (compactication level)

How compact should tags be rendered, as specified in Document → View → Compactification. Possible values are `none`, `inline`, `normal`, `inline tags` and `all`.

src-close := `compact` (closing style for long tags)

The rendering style of closing tags as specified in Document → View → Closing style. Possible values are `repeat`, `long`, `compact` and `minimal`.

13.8. Miscellaneous environment variables

The following miscellaneous environment variables are mainly intended for internal use:

save-aux := `true` (save auxiliary content)

This flag specifies whether auxiliary content has to be saved along with the document.

sfactor := `5` (shrinking factor)

The shrinking factor which is used for rendering.

par-no-first := `false` (disable first indentation for next paragraph?)

This flag disables first indentation for the next paragraph.

cell-format (current cell format)

This variable us used during the typsetting of tables in order to store the with-settings which apply to the current cell.

atom-decorations
line-decorations
page-decorations
xoff-decorations
yoff-decorations (auxiliary variables for decorations)

These environment variables store auxiliary information during the typsetting of decorations.

Chapter 14

Built-in TeXmacs primitives

In this chapter, we describe those built-in TeXmacs primitives which are intended to be used in normal documents. The additional primitives which are used for writing style files are described in a separate chapter.

14.1. Fundamental primitives

⟨document|*par-1*|···|*par-n*⟩ (vertical sequence of paragraphs)

This primitive is used for sequences of logical paragraphs. A simple, plain text document is made of a sequence of paragraphs. For instance,

> A simple document.
>
> Made of several paragraphs. The second paragraph is very long, so that it is hyphenated across several line.

is internally represented as a document with two subtrees:

> ⟨document|
> A simple document.|
> Made of several paragraphs. The second paragraph is very long, so that it is hyphenated across several line.⟩

From the visual point of view, different paragraphs are often separated by some vertical whitespace. Alternatively, new paragraphs are indicated through the use of an additional indentation. The root of a TeXmacs document is usually a document node.

The document tag is also used for marking multi-paragraph content inside other tags, such lists or theorem-like environments. Environments which require the use of a document tag for at least one argument are called "block environments".

⟨paragraph|*unit-1*|···|*unit-n*⟩ (vertical sequence of paragraph units)

This not yet implemented primitive is a variant of document. While a document is made up of logical paragraphs, a paragraph is made up of "paragraph units". From a visual point of view, different paragraphs are singled out using some additional space or indentation. New paragraph units rather correspond to simple new lines. Typically, displayed equations are also paragraph units in a larger paragraph.

⟨concat|*item-1*|···|*item-n*⟩ (horizontal sequence of inline markup)

This primitive is used for sequences of line items, also called "inline content". For instance, Some *emphasized* text.

is internally represented as:

⟨concat|Some |⟨em|emphasized⟩| text.⟩

The concat operator is essential to put compound structures in trees taking multiple parameters. For example, let us place the previous fragment in a multi-paragraph context:

Multiple paragraphs.

Some *emphasized* text.

In this example, we need the concat tag in order to indicate that “Some *emphasized* text.” correponds to a single paragraph:

```
⟨document|
    A simple document.|
    ⟨concat|Some |⟨em|emphasized⟩| text.⟩⟩
```

Notice that block tags like document may contain inline tags such as concat as its children, but not *vice versa*. In order to typeset line content before or after block content, one has to use the surround tag below.

⟨surround|*left*|*right*|*body*⟩ (surround block content with inline content)

Although it is not possible in $\mathrm{T_EX_{MACS}}$ to use block content inside horizontal concatenations, it is sometimes useful to add some additional inline content before or after a block environment. The surround primitive serves this purpose, by adding a *left* and *right* surrounding to some block content *body*. For instance,

```
⟨surround|↯ ||
    ⟨theorem|
        Given P ∈ 𝕋{F} and f < g ∈ 𝕋 with P(f) P(g) < 0, there exists
        an h ∈ 𝕋 with P(h) = 0.⟩⟩
```

produces

↯ THEOREM 14.1. *Given* $P\in\mathbb{T}\{F\}$ *and* $f<g\in\mathbb{T}$ *with* $P(f)\,P(g)<0$*, there exists an* $h\in\mathbb{T}$ *with* $P(h)=0$.

In general, the surround is mainly used in style files, but it occasionally turns out to be useful in regular documents as well.

14.2. FORMATTING PRIMITIVES

14.2.1. White space primitives

⟨vspace|*len*⟩

⟨vspace|*len*|*min*|*max*⟩ (vertical space after)

This primitive inserts an elastic vertical space after the current paragraph. All operands must be length values. The *len* argument specifies the default length and the *min* and *max* arguments the bounds to vertical stretching for page breaking and filling. If *min* and *max* are not specified, then they are determined implicitly from the length unit of *len*.

Notice that operands are not evaluated, so they must be literal strings.

⟨vspace*|*len*⟩
⟨vspace*|*len*|*min*|*max*⟩ (vertical space before)

This primitive is similar to vspace, except that the vertical space is inserted *before* the current paragraph. The actual vertical space between two consecutive paragraphs is the *maximum*, not the sum, of the vertical spaces specified by the the vspace and vspace* tags in the surrounding paragraphs.

⟨space|*len*⟩
⟨space|*len*|*bot*|*top*⟩ (rigid horizontal space)

This primitive inserts an empty box whose width is *len*, and whose bottom and top sides are at distances *bot* and *top* from the baseline.

If *bot* and *top* are not specified, then an empty box is inserted whose bottom is on the baseline and whose height is the same as the lowercase letter x in the current font.

Notice that operands are not evaluated, so they must be literal strings.

⟨hspace|*len*⟩
⟨hspace|*len*|*min*|*max*⟩ (stretchable horizontal space)

This primitive inserts inserts a stretchable horizontal space of nominal width *len*, which must be a length value. The *min* and *max* arguments specify bounds to horizontal stretching for line breaking and filling. If *min* and *max* are not specified, then they are determined implicitly from the length unit of *len*.

Notice that operands are not evaluated, so they must be literal strings.

⟨htab|*min*⟩
⟨htab|*min*|*weight*⟩ (horizontal spring)

Springs are horizontal spaces which extend so the containing paragraph takes all the available horizontal space. When a paragraph is line wrapped, split in several visual lines, only springs in the last line are extended.

A spring has a *minimal width* and a *weight*. If the weight is 0, the spring is *weak*, otherwise it is *strong*. If a line contains mixed weak and strong springs, only the strong springs extend.

The fraction of the available horizontal space taken up by each strong spring is proportional to its weight. If there are only weak springs, they share the available space evenly.

> ⟨htab|*min*⟩ inserts a strong spring of minimal width *min* and of weight unity. The *min* operand must be a length value.
>
> ⟨htab|*min*|*weight*⟩ specifies the weight, which can be a positive decimal number or one of the two special values documented below.
>
> ⟨htab | *min* | first⟩ inserts a *tail weak* spring, only the first one in a paragraph is significant.

⟨htab | *min* | last⟩ inserts a *head weak* spring, only the last one in a paragraph is significant.

Operands are not evaluated and must be literal strings.

Weak springs are useful in style-sheets. For example, tail weak springs are used to make the list environment extend to across the full paragraph, so vertical motion commands in nested lists behave as expected. In regular documents, springs are often used to place some text on the right side of the page and some other text on the left side.

14.2.2. Line breaking primitives

A simple document is a sequence of *logical paragraphs*, one for each subtree of a document or paragraph node. Paragraphs whose width exceed the available horizontal space are broken into *physical lines* by the hyphenation algorithm. By default, hyphenated lines are justified: horizontal spaces can be shrunk or extended in order to produce a good-looking layout.

⟨new-line⟩ (start a new paragraph)

This is a deprecated tag in order to split a logical paragraph into several logical paragraphs without creating explicit subtrees for all paragraphs.

We recall that logical paragraphs are important structures for the typesetting process. Many primitives and environment variables (vertical spacing, paragraph style, indentation, page breaking, etc.) operate on whole paragraphs or at the boundaries of the enclosing paragraph.

⟨next-line⟩ (start a new line)

This is a tag which will become deprecated as soon as the paragraph primitive will be correctly implemented. Its usage is similar to the new-line tag with the difference that we start a new logical paragraph unit instead of a new logical paragraph.

Currently, the next-line tag can also be used in order to force a line break with the addional property that the line before the break is not justified or filled.

⟨line-break⟩ (line breaking hint, with filling)

Print an invisible space with zero hyphenation penalty. The line breaking algorithm searches for the set of hyphenation points minimizing the total penalty, so line breaking is much more likely to occur at a line-break than anywhere else in its vicinity.

Unlike next-line, this is a hint which may or may not be obeyed by the typesetter, and it does not prevent the previous line from being filled.

⟨no-break⟩ (forbid line breaking at this point)

Set an hyphenation point with an infinite penalty. That is useful when the hyphenation patterns for a language fall short of preventing some forbidden patterns like "arse-nal" or "con-genital". An alternative way to prevent breaks is to use the group tag.

14.2.3. Indentation primitives

There are two main ways to distinguish between successive paragraphs: separate them by a small vertical space, or use an indentation for each new paragraph. The indentation can be explicitly controlled using the no-indent, yes-indent, no-indent* and yes-indent* tags. The no-indent and yes-indent primitives apply to the current paragraph, while the no-indent* and yes-indent* apply the next paragraph.

⟨no-indent⟩
⟨yes-indent⟩

Disable or enable indentation for the current paragraph. For instance, the code

> ⟨no-indent⟩This is a long paragraph which demonstrates the disabling indentation using the ⟨markup|no-indent⟩ primitive.
> ⟨yes-indent⟩This is a long paragraph which demonstrates enabling indentation using the ⟨markup|yes-indent⟩ primitive.

typically produces

> This is a long paragraph which demonstrates the disabling indentation using the no-indent primitive.
>
> This is a long paragraph which demonstrates enabling indentation using the yes-indent primitive.

⟨no-indent*⟩
⟨yes-indent*⟩

Disable or enable indentation for the next paragraph. For instance,

> A first paragraph.⟨yes-indent*⟩
> A second paragraph.

typically produces

> A first paragraph.
>
> A second paragraph.

Notice that no-indent and yes-indent override no-indent* and yes-indent* directives in the previous paragraph.

Currently, the no-indent* and yes-indent* tags are mainly used in order to control the indentation after section titles or environments like equation which usually correspond to paragraph units. In the future, when sectional tags will take the section bodies as arguments, and when the paragraph tag will be correctly implemented, the no-indent* and yes-indent* will become deprecated.

14.2.4. Page breaking primitives

The physical lines in a document are broken into pages in a way similar to how paragraphs are hyphenated into lines. The page breaker performs *page filling*, it tries to distribute page items evenly so text runs to the bottom of every page. It also tries to avoid *orphans and widows*, which are single or pairs of soft lines separated from the rest of their paragraph by a page break, but these can be produced when there is no better solution.

⟨no-page-break⟩ (prevent automatic page breaking after this line)

Prevent the occurrence of an automatic page break after the current line. Set an infinite page breaking penalty for the current line, similarly to no-break.

Forbidden page breaking points are overridden by “new page” and “page break” primitives.

⟨no-page-break*⟩ (prevent automatic page breaking before this line)

Similar to no-page-break, but set the page breaking penalty of the previous line.

⟨new-page⟩ (start a new page after this line)

Cause the next line to appear on a new page, without filling the current page. The page breaker will not try to position the current line at the bottom of the page.

⟨new-page*⟩ (start a new page before this line)

Similar to new-page, but start the new page before the current line. This directive is appropriate to use in chapter headings.

⟨page-break⟩ (force a page break after this line)

Force a page break after the current line. A forced page break is different from a new page, the page breaker will try to position the current line at the bottom of the page.

Use only to fine-tune the automatic page breaking. Ideally, this should be a hint similar to line-break, but this is implemented as a directive, use only with extreme caution.

⟨page-break*⟩ (force a page break before this line)

Similar to page-break, but force a page break before the current line.

When several “new page” and “page break” directives apply to the same point in the document, only the first one is effective. Any new-page or page-break after the first one in a line is ignored. Any new-page or page-break in a line overrides any new-page* or page-break* in the following line. Any new-page* or page-break* after the first one in a line is ignored.

14.2.5. Box operation primitives

⟨move|*content*|*delta-x*|*delta-y*⟩ (adjust position)

This primitive moves the box with the specified *content* by *delta-x* to the right and *delta-y* upwards. It may be used for fine-grained positioning.

⟨resize|*content*|*left-lim*|*bot-lim*|*right-lim*|*top-lim*⟩ (adjust size)

Resize the box for the *content* according to new left, bottom, right and top limits *left-lim*, *bot-lim*, *right-lim* and *top-lim*. The limits may be either be empty strings (in which case the old limit is taken), an absolute coordinate, or a limit computed as a function of the old limit.

In the last case, the limit string should be of the form <pos><op><len>. The first character <pos> indicates a position in the old box and should be either `l` (left), `b` (bottom), `c` (center), `r` (right) or `t` (top). The second character <op> indicates the operation which will be performed on this position and the remaining length string <len> in order to yield the new position. Possible operations are `+`, `-`, `[` and `]`. The brackets `[` and `]` stand for “minimum” and “maximum”. For instance, the code

```
⟨⟨resize|Hopsa|l-5mm||r+5mm||⟩⟩
```

widens the box for “Hopsa” by `5mm` on each side:

```
(    Hopsa    )
```

⟨if*|*condition*|*content*⟩ (conditional appearance of box)

The box with the *content* is displayed as usual if the *condition* is satisfied and displayed as whitespace otherwise. This primitive is used in particular for the definition of the phantom macro. For instance, the non-text "   " is produced using ⟨if* | false | phantom⟩.

⟨repeat|*content*|*pattern*⟩ (fill line)

This primitive can be used to decorate some *content* with a given *pattern*. For instance, when defining the macro

```
⟨assign|wipe-out|⟨macro|x|⟨repeat|x|⟨with|color|red|/⟩⟩⟩⟩
```

the code ⟨wipe-out|obsolete⟩ produces obsolete. The repeat primitive may also be used to fill the current line with a given content, like the dots in tables of contents.

⟨datoms|*foo*|*content*⟩
⟨dlines|*foo*|*content*⟩
⟨dpages|*foo*|*content*⟩ (decorations)

These primitives are used to decorate *a posteriori* the lines of a paragraph, the lines of a page, or the pages of a document. Currently, only decorations of atoms on lines of a paragraph have been implemented.

The first argument *foo* is a macro which will be applied to all boxes in the line and the second argument *content* is the part of the paragraph to which the decoration will be applied. For instance, the construction

```
⟨datoms|
    ⟨macro|x|[x]⟩|
    body⟩
```

may be used in order to visualize the boxes in a given paragraph:

Here is a sufficiently long paragraph. Here is a sufficiently long paragraph.
Here is a sufficiently long paragraph. Here is a sufficiently long paragraph.
Here is a sufficiently long paragraph. Here is a sufficiently long paragraph.

When used in combination with the repeat primitive, one may for instance produce the dotted lines in tables of contents using the macro

```
⟨assign|
    toc-dots|
    ⟨macro|
        ⟨datoms|
            ⟨macro|x|⟨repeat|x|⟨space|0.2fn⟩.⟨space|0.2fn⟩⟩⟩|
            ⟨htab|5mm⟩⟩⟩⟩
```

Notice that the datoms primitive is quite fragile, because the *foo* macro has no access to the environment in which *content* is typeset.

14.3. Mathematical primitives

⟨left|*large-delimiter*⟩
⟨left|*large-delimiter*|*size*⟩
⟨left|*large-delimiter*|*bottom*|*top*⟩
⟨mid|*large-delimiter*|⋯⟩
⟨right|*large-delimiter*|⋯⟩ (large delimiters)

These primitives are used for producing large delimiters, like in the formula

$$\left\langle \frac{1}{a_1} \middle| \frac{1}{a_2} \middle| \cdots \middle| \frac{1}{a_n} \right\rangle.$$

Matching left and right delimiters are automatically sized so as contain the enclosed expression. Between matching left and right delimiters, the formula may contain an arbitrary number of middle delimiters, which are sized in a similar way. Contrary to TEX, the depth of a large delimiter is not necessarily equal to its height, so as to correctly render formulas like

$$f\left(\frac{1}{x + \frac{1}{y + \frac{1}{z}}} \right)$$

The user may override the automatically determined size by specifying additional length parameters *size* or *bottom* and *top*. For instance,

```
f⟨left|(|-8mm|4mm⟩x⟨mid|||8mm⟩y⟨right|)|-4mm|8mm⟩
```

is rendered as

$$f\left(x \middle| y \right)$$

The *size* may also be a number n, in which case the n-th available size for the delimiter is taken. For instance,

```
g⟨left|(|0⟩⟨left|(|1⟩⟨left|(|2⟩⟨left|(|3⟩z⟨right|)|3⟩⟨right|)|2⟩⟨right|)|1⟩⟨right|
)|0⟩
```

is rendered as

$$g(\big(\Big(\bigg(z\bigg)\Big)\big))$$

⟨big|*big-symbol*⟩ (big symbols)

This primitive is used in order to produce big operators as in

$$\sum_{i=0}^{\infty} a_i z^i \tag{14.1}$$

The size of the operator depends on whether the formula is rendered in "display style" or not. Formulas in separate equations, like (14.1), are said to be rendered in display style, contrary to formulas which occur in the main text, like $\sum_{i=0}^{\infty} a_i z^i$. The user may use Format → Display style to override the current settings.

Notice that the formula (14.1) is internally represented as

⟨big|sum⟩⟨rsub|i=0⟩⟨rsup|<infty>⟩a⟨rsub|i⟩*z⟨rsup|i⟩⟨big|.⟩

The invisible big operator ⟨big|.⟩ is used to indicate the end of the scope of ⟨big|sum⟩.

⟨frac|*num*|*den*⟩ (fractions)

The frac primitive is used in order to render fractions like $\frac{x}{y}$. In display style, the numerator *num* and denominator *den* are rendered in the normal size, but display style is turned of when typesetting *num* and *den*. When the display style is turned of, then the arguments are rendered in script size. For instance, the content

⟨frac|1|a⟨rsub|0⟩+⟨frac|1|a⟨rsub|1⟩+⟨frac|1|a⟨rsub|2⟩+<ddots>⟩⟩⟩

is rendered in display style as

$$\frac{1}{a_0 + \frac{1}{a_1 + \frac{1}{a_2 + \ddots}}}$$

⟨sqrt|*content*⟩
⟨sqrt|*content*|*n*⟩ (roots)

The sqrt primitive is used in order to render square roots like $\sqrt{x}$ or *n*-th roots like $\sqrt[3]{x}$. The root symbol is automatically sized so as to encapsulate the *content*:

$$\sqrt[i+j]{\frac{f(x)}{y^2+z^2}}$$

⟨lsub|*script*⟩
⟨lsup|*script*⟩
⟨rsub|*script*⟩
⟨rsup|*script*⟩ (scripts)

These primitives are used in order to attach a *script* to the preceeding box in a horizontal concatenation (in the case of right scripts) or the next one (in the case of left scripts). When there is no such box, then the script is attached to an empty box. Moreover, when both a subscript and a superscript are specified on the same side, then they are merged together. For instance, the expression

⟨rsub|a⟩⟨rsup|b⟩+⟨lsub|1⟩⟨lsup|2⟩x⟨rsub|3⟩⟨rsup|4⟩=y⟨rsub|1⟩+⟨lsub|c⟩

is rendered as

$${}_a^b + {}_1^2x_3^4 = y_1 + {}_c$$

When a right script is attached to an operator (or symbol) which accepts limits, then it is rendered below or above instead of beside the operator:

$$\lim_{n \to \infty} a_n$$

Scripts are rendered in a smaller font in non-display style. Nevertheless, in order to keep formulas readable, the size is not reduced below script-script-size.

⟨lprime|*prime-symbols*⟩
⟨rprime|*prime-symbols*⟩ (primes)

Left and right primes are similar to left and right superscripts, except that they behave in a different way when being edited. For instance, when your cursor is behind the prime symbol in f' and you press backspace, then the prime is removed. If you are behind f^n and you press backspace several times, then you first enter the superscript, next remove n and finally remove the superscript. Notice also that *prime-symbols* is necessarily a string of concatenated prime symbols. For instance, $f'^{\dagger}$ is represented by f⟨rprime|'<dag>⟩.

⟨below|*content*|*script*⟩
⟨above|*content*|*script*⟩ (scripts above and below)

The below and above tags are used to explicitly attach a *script* below or above a given *content*. Both can be mixed in order to produce content with both a script below and above:

$$\overset{\infty}{\underset{i=1}{\mathrm{xor}}}\, x_i$$

can be produced using

⟨above|⟨below|xor|i=1⟩|<infty>⟩ x⟨rsub|i⟩

⟨wide|*content*|*wide-symbol*⟩
⟨wide*|*content*|*wide-symbol*⟩ (wide symbols)

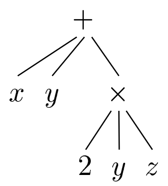
These primitives can be used in order to produce wide accents above or below some mathematical *content*. For instance $\overline{x+y}$ corresponds to the markup ⟨wide | x+y | <bar>⟩.

⟨neg|*content*⟩ (negations)

This primitive is mainly used for producing negated symbols or expressions, such as $\not\rightarrowtail$ or $\not{d}$.

⟨tree|*root*|*child-1*|⋯|*child-n*⟩ (trees)

This primitive is used to produce a tree with a given *root* and children *child-1* until *child-n*. The primitive should be used recursively in order to produce trees. For instance,



corresponds to the markup

⟨tree|+|x|y|⟨tree|<times>|2|y|z⟩⟩

In the future, we plan to provide further style parameters in order to control the rendering.

14.4. TABLE PRIMITIVES

Tables are always present in documents inside evaluable tags which take a tformat operand. All fundamental table structures have inaccessible borders. The basic top-level table tag is tabular.

⟨tformat|*with-1*|⋯|*with-n*|*table*⟩ (table formatting container)

Every tabular structure in a document contains a tformat tag.

⟨tformat|*table*⟩ means the table and cell variables defined in the top-level table tag are not modified. The *table* argument may be a table or a nested tformat tag, the latter does not appear in documents but is produced by the evaluation of the top-level tag.

⟨tformat|*with-1*|⋯|*with-n*|*table*⟩ is used when the table contains specific formatting information. The *with-1* to *with-n* arguments must all be twith or cwith tags.

⟨twith|*var*|*val*⟩ (set a table variable)

The formatting of the table as a whole is specified by a number of *table variables*, which are used internally and do not appear in the environment like regular typesetter variables.

The twith primitive sets the table variable *var* (literal string) to the value *val* (evaluated).

⟨cwith|*top-row*|*bot-row*|*left-col*|*right-col*|*var*|*val*⟩ (set a cell variable for a range)

The formatting of cells is specified by a number of *cell variables*, which are used internally and do not appear in the environment like regular typesetter variables. Rows, columns, and generally any rectangular range of cells can associated to a cell variable setting by a single cwith tag.

The cwith primitive sets the cell variable *var* (literal string) to the value *val* (evaluated) for the range of cells spanning rows *top-row* to *bot-row* and columns *left-col* to *right-col* (literal non-zero integers).

Range coordinates must be non-zero literal integers, positive values are counted left to right and top to bottom, negative values are counted right to left and bottom to top. For example, 2 means the second row or column and -1 means the last row or column.

Typical values for (*top-row*, *bot-row*, *left-col*, *right-col*) are $(r, r, 1, -1)$ for "row r", $(1, -1, c, c)$ for "column c", and (r, r, c, c) for "the cell at row r, column c". When new cells are inserted, it makes a difference whether the rows are counted from the top or bottom, and the columns are counted from the left or right. If m is the number of rows and n the number of columns, then r and $r - m - 1$ represent the same row—the former is relative to the top border while the latter is relative the bottom border. Similarly, c and $c - n - 1$ represent the same column.

⟨table|*row-1*|⋯|*row-n*⟩ (row container)

The only purpose of the table tag is to contain row tags. The number of rows in a table is the number of subtrees in its table tag.

⟨row|*cell-1*|⋯|*cell-k*⟩ (cell container)

The only purpose of the row tag is to contain cell tags. All row tags in a given table must have exactly as many subtrees, all cell tags, as there are columns in the table.

⟨cell|*content*⟩ (cell data container)

Table cells can contain any document fragment. A cell may directly contain an inline content tag or a concat, if it has block content it must always contain a document tree.

A cell whose operand is a document is a *multi-paragraph cell*. Since tables are allowed in line context, this is the only construct which allows, indirectly, the nesting of a block context within a line context. Note that most block content can only be typeset correctly within an hyphenated cell, this is controlled by the *cell-hyphen* table variable.

⟨subtable|*table*⟩ (subtable cell data)

In addition to regular markup, cells can accept subtable as an operand. The operand of subtable is a tformat tree containing regular table data.

A similar effect can be obtained with normal table by setting the cell's padding to zero in all directions, the extra twist of a subtable is its inaccessible border positions.

⟨tmarker|*table*⟩ (decoration origin marker)

This tag is used in the definition of cell decorations, see the documentation of the *cell-decoration* environment variable.

It is also used outside tables, in the switch tag to mark the currently displayed position.

⟨tabular|*table*⟩ (built-in tabular macro)

This macro implements standard left aligned tables without borders. Although the tabular macro is built-in into TeXmacs, it should not really be considered as a primitive. However, it is not part of any style file either.

14.5. Linking primitives

⟨label|*name*⟩ (reference target)

The operand must evaluate to a literal string, it is used as a target name which can be referred to by reference, pageref and hlink tags.

Label names should be unique in a document and in a project.

Examples in this section will make references to an example label named “there”.

⟨label|there⟩

⟨reference|*name*⟩ (reference to a name)

The operand must evaluate to a literal string, which is the name of a label defined in the current document or in another document of the current project.

⟨reference|there⟩

The reference is typeset as the value of the variable *the-label* at the point of the target label. The *the-label* variable is set by many numbered structures: sections, figures, numbered equations, etc.

A reference reacts to mouse clicks as an hyperlink.

⟨pageref|*name*⟩ (page reference to a name)

The operand must evaluate to a literal string, which is the name of a label defined in the current document or in another document of the current project.

⟨pageref|there⟩

The pageref is typeset as the number of the page containing the target label. Note that page numbers are only computed when the document is typeset with page-breaking, that is not in "automatic" or "papyrus" page type.

A pageref reacts to mouse clicks as an hyperlink.

⟨hlink|*content*|*url*⟩ (inline hyperlink)

This primitive produces an hyperlink with the visible text *content* pointing to *url*. The *content* is typeset as inline *url*. The *url* must evaluate to a literal string in URL syntax and can point to local or remote documents, positions inside documents can be be specified with labels.

The following examples are typeset as hyperlinks pointing to the label "there", respectively in the same document, in a document in the same directory, and on the web.

⟨hlink|same document|`#there`⟩

⟨hlink|same directory|`file.tm#there`⟩

⟨hlink|on the web|`http://example.org/#there`⟩

If the document is not editable, the hyperlink is traversed by a simple click, if the document is editable, a double-click is required.

⟨include|*url*⟩ (include another document)

The operand must be a literal string and is interpreted as a file name. The content of this file is typeset in place of the include tag, which must be placed in block context.

⟨action|*content*|*script*⟩ (attach an action to content)

Bind a Scheme *script* to a double mouse click on *content*. For instance, when clicking here, you may launch an `xterm`. This action is encoded by ⟨action|here|`(system "xterm &")`⟩.

When clicking on actions, the user is usually prompted for confirmation, so as to avoid security problems. The user may control the desired level of security in Edit → Preferences → Security. Programmers may also declare certain Scheme routines to be "secure". Scheme programs which only use secure routines are executed without confirmation from the user.

⟨mutator|*content*|*script*⟩ (a tag which may modify itself)

The *content* of a mutator tag is automatically determined by the Scheme *script*. More precisely, TeXmacs periodically determines all mutator tags which are present in the currently opened. For each such tag, it sets the *mutator path* to the path for accessing its *content* and calls the Scheme *script*. This script is allowed to modify the content of the mutator and even other parts of the document (for efficiency reasons it is nevertheless recommended to mainly modify the content itself). In order to retrieve the mutator path from within Scheme, one should use the command `(get-mutator-path)`.

Mutators are very useful in situations where TEXMACS communicates with extern systems in an interactive way. For instance, the current implementation of computer algebra (and other) sessions uses mutators. This allows the user to continue typing elsewhere in the document while the extern system is computing. Since mutator tags are automatically localized by the editor, the behaviour remains correct when the position of the mutator changes during the computation (this happens for instance when inserting a new paragraph at the start of your document).

14.6. MISCELLANEOUS PHYSICAL MARKUP

⟨group|*content*⟩ (atomic entity)

Typeset the *content*, which must be line content, as an atomic line item. Hyphenation within the group and special spacing handling on its borders are disabled.

⟨float|*type*|*where*|*body*⟩ (floating page insertion)

Floating insertions are page items which are typeset "out of band", they are associated to two boxes: the anchor box marks the structural position of the float, the floating box contains the typeset *body* operand. This facility is used by footnotes and floating blocks.

The first and second operands are evaluated, but for clarity the first operand appears as a literal string in the examples. Since the *body* is typeset out of band, it may be block content even if the float occurs in line context.

⟨float|footnote||*body*⟩ produces a footnote insertion, this should only be used within the footnote macro and is considered style markup. The floating box of a footnote is typeset at the end of the the page containing the anchor box.

⟨float | float | *where* | *body*⟩ produces a floating block, this is considered physical markup. The position of the floating box is chosen by the page breaker, which uses this extra freedom to minimize the page breaking penalty.

The *where* operand must evaluate to a string which may contain the following characters:

t. Allow the floating box at page *top*.

b. Allow the floating box at page *bottom*.

h. Allow the floating box "*here*", in the middle of the page near the anchor box.

f. *Force* the floating box within the same page as the anchor box.

⟨specific|*medium*|*body*⟩ (medium-specific content)

This primitive marks *body* for output only on the specified *medium*. The following values of *medium* are supported:

texmacs. The *body* is typeset as usual line content.

latex. The *body*, which must be a string, is not visible from within TEXMACS, but it will be included in a verbatim way when the document is exported to LATEX.

html. Similar to the `latex` medium, but for HTML exports.

screen. The *body* is only typeset when the document is visualized on a screen. This may be useful to provide additional visual information to the user during the editing phase which should disappear when printing out. A similar tag which may be used for this purpose is flag.

printer. This medium is complementary to `screen`, when the *body* should only be visible when printing out, but not when the document is displayed on the screen.

⟨raw-data|*data*⟩ (binary content)

In some contexts you need to embed uneditable data inside a document, most of the time this is uneditable binary data. The raw-data primitive makes it impossible to view or modify its subtree from within the editor.

Chapter 15

Primitives for writing style files

15.1. Environment primitives

The current environment both defines all style parameters which affect the typesetting process and all additional macros provided by the user and the current style. The primitives in this section are used to access and modify environment variables.

⟨assign|*var*|*val*⟩ (variable mutation)

This primitive sets the environment variable named *var* (string value) to the value of the *val* expression. This primitive is used to make non-scoped changes to the environment, like defining markup or increasing counters.

This primitive affects the evaluation process —through value, provides, and macro definitions— and the typesetting process —through special typesetter variables.

Example 15.1. Enabling page breaking by style.

The `page-medium` is used to enable page breaking. Since only the initial environment value for this variable is effective, this assignation must occur in a style file, not within a document.

⟨assign|*page-medium*|paper⟩

Example 15.2. Setting the chapter counter.

The following snippet will cause the immediately following chapter to be number 3. This is useful to get the the numbering right in `book` style when working with projects and include.

⟨assign|*chapter-nr*|2⟩

The operand must be a literal string and is interpreted as a file name. The content of this file is typeset in place of the include tag, which must be placed in block context.

⟨with|*var-1*|*val-1*|⋯|*var-n*|*val-n*|*body*⟩ (variable scope)

This primitive temporarily sets the environment variables *var-1* until *var-n* (in this order) to the evaluated values of *val-1* until *val-n* and typesets *body* in this modified environment. All non-scoped change done with assign to *var-1* until *var-n* within *body* are reverted at the end of the with.

This primitive is used extensively in style files to modify the typesetter environment. For example to locally set the text font, the paragraph style, or the mode for mathematics.

⟨value|*var*⟩ (variable value)

This primitive evaluates the current value of the environment variable *var* (literal string). This is useful to display counters and generally to implement environment-sensitive behavior.

This primitive is used extensively in style files to modify the typesetter environment. For example to locally set the text font, the paragraph style, or the mode for mathematics.

⟨provides|*var*⟩ (definition predicate)

This predicate evaluates to `true` if the environment variable *var* (string value) is defined, and to `false` otherwise.

That is useful for modular markup, like the session environments, to fall back to a default appearance when a required package is not used in the document.

15.2. MACRO PRIMITIVES

Macros can be used to define new tags and to build procedural abstractions in style files.

Older versions of $\mathrm{T_EX_{MACS}}$ used to make a distinction between macros (all children accessible) and functions (no accessible child). In modern $\mathrm{T_EX_{MACS}}$ there are only macros: the accessibility of children is determined heuristically and can be controlled with drd-props.

⟨macro|*var-1*|⋯|*var-n*|*body*⟩ (macro of fixed arity)

This primitive returns a macro (the $\mathrm{T_EX_{MACS}}$ analogue of a λ-expression) with n arguments, named after the literal strings *var-1* until *var-n*.

New tags are defined by storing macros in the environment. Most of the time, macros are stored without scope with assign, but it is sometimes useful to redefine a tag locally within the scope of a with. For example, itemized and enumerated environment redefine item locally.

EXAMPLE 15.3. Definition of the abbr tag

⟨assign|*abbr*|⟨macro|x|⟨group|x⟩⟩⟩

Storing a macro in the environment defines a tag whose arity is fixed to the number of arguments taken by the macro.

⟨arg|*var*|*index-1*|⋯|*index-n*⟩ (retrieve macro arguments)

This primitive is used to retrieve the arguments of a macro within its body. For instance, ⟨arg | *var*⟩ expands the content of the macro argument with name *arg* (literal string). Of course, this argument must be defined by a macro containing the arg tag.

This tag is similar to value, but differs in important ways:

- The argument namespace is distinct from the environment, ⟨arg|*var*⟩ and ⟨value|*var*⟩ will generally evaluate to different values (although you should not rely on this).

- The value of arg retains the position of the macro argument in the document tree, that makes it possible to edit the arguments of a macro-defined tag while it is active.

When more than one arguments are specified, ⟨arg|*var*|*index-1*|⋯|*index-n*⟩ expands to a subtree of the argument *var*. The value of the named argument must be a compound tree (not a string). The operands *var* until *index-n* must all evaluate to positive integers and give the path to the subtree of the macro argument.

⟨xmacro|*var*|*body*⟩ (macro with a variable arity)

This primitive returns a macro (the $\mathrm{T_EX_{MACS}}$ analogue of a λ-expression) capable of taking any number of arguments. The arguments are stored in the macro variable with name *var* (a literal string) during the evaluation of the *body*. The i-th individual argument can then be accessed using ⟨arg|*var*|*i*⟩.

⟨map-args|*foo*|*root*|*var*⟩
⟨map-args|*foo*|*root*|*var*|*first*⟩
⟨map-args|*foo*|*root*|*var*|*first*|*last*⟩ (map a tag on subtrees of an argument)

This primitive evaluates to a tree whose root is labeled by *root* and whose children are the result of applying the macro *foo* to the children of the macro argument with name *var*.

By default, the macro *foo* is applied to all children. If *first* has been specified, then we rather start at the i-th child of *var*, where i is the result of evaluating *first*. If *last* has been specified to, then we stop at the j-th child of *var* (the j-th child not being included), where j is the result of evaluating *last*. In this last case, the arity of the returned tree is therefore $j-i$.

Stated otherwise, map-args applies *foo* to all subtrees of the macro argument *var* (or a range of subtrees if *first* and *last* are specified) and collect the result in a tree with label *root*. In addition, the second argument to *foo* gives its position of the first argument in the expansion of *var*.

The map-args is analogue to the Scheme function `map`. Since $\mathrm{T_EX_{MACS}}$ use labeled trees, the label of the mapping list must also be specified.

Example 15.4. Comma-separated lists.

The comma-separated tag has any arity (though it does not make much sense with arity zero) and typeset its operands interspersed with commas.

```
⟨assign|comma-extra|⟨macro|x|, x⟩⟩
⟨assign|comma-separated|
    ⟨xmacro|args|
        ⟨concat|
            ⟨arg|args|0⟩|
            ⟨map-args|comma-extra|concat|args|1⟩⟩⟩⟩
```

⟨eval-args|*var*⟩ (macro with a variable arity)

This primitive evaluates to the tree with the same label as the expansion of the argument *var* and whose subtrees are the result of the evaluation of the subtrees of the expansion of *var*.

⟨compound|*foo*|*arg-1*|···|*arg-n*⟩ (expand an unnamed macro)

This primitive is useful to expand macros which are the result of a computation: it applies the macro which is the result of the evaluation of *foo* to the arguments *arg-1* until *arg-n*. The compound primitive is useful in call-back and lambda programming idioms, where a *higher-level macro* is given a macro as an operand, which it may later apply under certain conditions or with operands which are not known the client code.

Actually, in the current implementation, *foo* may either evaluate to a macro or to a literal string which gives the name of a macro. However, we discourage users to rely on the second case.

EXAMPLE 15.5. Lambda programming with macros.

In the code below, ⟨filter | *pred* | *t*⟩ expects a macro *pred* and a tuple *t* on input and returns a tuple containing the elements of *t* for which *pred* evaluates to `true`.

```
⟨assign|filter|
  ⟨macro|pred|t|
    ⟨if|
      ⟨equal|⟨length|t⟩|0⟩|
      ⟨tuple⟩|
      ⟨merge|
        ⟨if|
          ⟨compound|pred|⟨look-up|t|0⟩⟩|
          ⟨tuple|⟨look-up|t|0⟩⟩|
          ⟨tuple⟩⟩|
        ⟨filter|pred|⟨range|t|1|⟨length|t⟩⟩⟩⟩⟩⟩⟩
```

As an application, we may define a macro ⟨evens | *t*⟩, which expects *t* to be a tuple containing integers, and which returns the tuple of integers in *t* which are divisible by 2.

```
⟨assign|evens|⟨macro|t|⟨filter|⟨macro|x|⟨equal|⟨mod|x|2⟩|0⟩⟩|t⟩⟩⟩
```

⟨drd-props|*var*|*prop-1*|*val-1*|···|*prop-n*|*val-n*⟩ (set D.R.D. properties of a tag)

The arity and children accessibility of tags defined by macros are determined heuristically by default. The drd-props primitive overrides this default for the environment variable (usually a macro) with name *var*. The currently supported property-value pairs are:

(arity, *n*) — Sets the arity to the given fixed value n (literal integer).

(accessible, all) — Make it impossible to deactivate the tag with normal editor actions. Inaccessible children become effectively uneditable.

(accessible, none) — Make it impossible to position the caret within the tag when it is active, so children can only be edited when the tag is inactive.

⟨get-label|*expression*⟩ (label of an expression)

Returns the label of the tree obtained when evaluating *expression*.

⟨get-arity|*expression*⟩ (arity of an expression)

Returns the label of the tree obtained when evaluating *expression*.

15.3. Flow control primitives

⟨if|*condition*|*if-body*⟩
⟨if|*condition*|*if-body*|*else-body*⟩ (conditional markup)

This primitive can be used to typeset *if-body* only if the *condition* is satisfied. If the optional *else-body* is specified, then it is typeset if and only if the *condition* fails.

Remark 15.6. It should be noticed that the use of conditional markup can be a bit tricky due to the fact that the accessability of arguments cannot necessarily be checked beforehand. For instance, in the macro definition

```
⟨macro|x|⟨if|⟨visibility-flag⟩|x⟩⟩
```

the macro argument x is accessible if and only if ⟨visibility-flag⟩ evaluates to true. This condition cannot necessarily be checked *a priori*. For certain editing operations, like searches or spell checking, the incorrect determination of the accessability may lead to the positioning of the cursor at unaccessible places, or to the ignorance of certain markup. In the future, we plan to improve this aspect of the editor, but it is better to avoid conditional markup whenever another solution can be found.

Remark 15.7. The conditional constructs are only fully implemented for inline markup. In the case when you need conditional markup for block structures you currently have to write macros for the if-case and the else-case and use the compound tag. For instance:

```
⟨assign|cold|⟨macro|x|⟨with|color|blue|x⟩⟩⟩
⟨assign|hot|⟨macro|x|⟨with|color|red|x⟩⟩⟩
⟨assign|adaptive|⟨macro|x|⟨compound|⟨if|⟨summer⟩|hot|cold⟩|x⟩⟩⟩
```

⟨case|*cond-1*|*body-1*|···|*cond-n*|*body-n*⟩
⟨case|*cond-1*|*body-1*|···|*cond-n*|*body-n*|*else-body*⟩ (case distinction)

These commands are respectively equivalent to

```
⟨if|cond-1|body-1|···⟨if|cond-n|body-n⟩⟩
⟨if|cond-1|body-1|···⟨if|cond-n|body-n|else-body⟩⟩
```

⟨while|*condition*|*body*⟩ (repeated evaluation)

This construct maybe used in order to repeatly execute a given *body* while a given *condition* is satisfied. For instance, when declaring

```
⟨assign|count|
    ⟨macro|from|to|
        ⟨with|i|from|
            ⟨concat|
                ⟨while|⟨less|i|to⟩|i, ⟨assign|i|⟨plus|i|1⟩⟩⟩|
                to⟩⟩⟩⟩
```

the code ⟨count|1|50⟩ produces

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50

15.4. EVALUATION CONTROL PRIMITIVES

This section describes several primitives for controlling the way expressions in the style-sheet language are evaluated. The primitives are analoguous to the SCHEME primitives `eval`, `quote`, `quasiquote`, etc., although the TEXMACS conventions are slightly different than those used by conventional functional languages like SCHEME.

⟨eval|*expr*⟩ (force evaluation)

Typeset the result of the evaluation of *expr*. This primitive is usually combined with a tag like quote or quasiquote for delaying the evaluation.

⟨quote|*expr*⟩ (delayed evaluation)

Evaluation of the expression ⟨quote | *expr*⟩ yields *expr* itself. This kind of delayed evaluation may be useful in combination with the eval primitive which forces evaluation.

⟨quasiquote|*expr*⟩ (delay evaluation and substitution)

This tag is a variant of the quote tag, which returns the expression *expr* in which all subexpressions of the form ⟨unquote|*subexpr*⟩ have been replaced by the evaluations of *subexpr*. For instance,

⟨assign | *hello* | ⟨quasiquote | ⟨macro | *name* | ⟨unquote | ⟨localize | Hello⟩⟩ *name*.⟩⟩⟩

may be used to define a macro hello whose value is localized for the current language. In a French document, the declaration would typically be equivalent to

⟨assign|*hello*|⟨macro|*name*|Bonjour *name*.⟩⟩

Notice however that it is usually better not to use the quasiquote primitive for such applications. When defining

⟨assign|*hello*|⟨macro|*name*|⟨localize|Hello⟩ *name*.⟩⟩

the typesetting of ⟨hello|Name⟩ would naturally adapt itself to the current language, while the above version would always use the language at the moment of the definition of the macro. Nevertheless, the first form does have the advantage that the localization of the word "Hello" only has to be computed once, when the macro is defined. Therefore, the quasiquote primitive may sometimes be used in order to improve performance.

⟨unquote|*subexpr*⟩ (mark substitutable subexpressions)

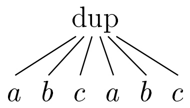
This tag is used in combination with quasiquote and quasi in order to mark the subexpressions which need to be evaluated.

⟨unquote*|*subexprs*⟩ (unquote splicing)

This tag is similar to unquote, except that the argument *subexprs* now evaluates to a list of subexpressions, which are inserted into the arguments of the parent node. For instance, consider the macro

```
⟨assign|fun|
    ⟨xmacro|x|
        ⟨quasi|
            ⟨tree | dup | ⟨unquote* | ⟨quote-arg | x⟩⟩ | ⟨unquote* | ⟨quote-arg |
            x⟩⟩⟩⟩⟩⟩
```

Then ⟨fun|a|b|c⟩ is typeset as



⟨quasi|*expr*⟩ (substitution)

This tag is a shortcut for ⟨eval|⟨quasiquote|*expr*⟩⟩. This primitive is often used in the TeXmacs style files in order to write macros which define sets of other macros. For instance, the macro

```
⟨assign|new-theorem|
    ⟨macro|name|text|
        ⟨quasi|
            ⟨assign|⟨unquote|name⟩|
                ⟨macro|body|
                    ⟨surround | ⟨no-indent⟩⟨strong | ⟨unquote | text⟩. ⟩ | ⟨right-
                    flush⟩|
                        body⟩⟩⟩⟩⟩⟩
```

may be used in order to define new theorem-like environments.

⟨quote-value|*var*⟩ (retrieve a value but don't evaluate)

When retrieving an environment variable *var*, one is usually interested in its typesetted value, as given by ⟨value|*var*⟩. In some cases, it may be useful to access the real, non-typesetted value. This can be done with ⟨quote-value|*var*⟩.

⟨quote-arg|*var*|*index-1*|⋯|*index-n*⟩ (retrieve an argument but don't evaluate)

When retrieving (a subexpression of) a macro argument *var*, one is usually interested in its typesetted value, as given by ⟨arg|*var*|*index-1*|⋯|*index-n*⟩. In some cases, it may be useful to access the real, non-typesetted value. This can be done with ⟨quote-arg|*var*|*index-1*|⋯|*index-n*⟩.

15.5. Functional operators

Funcational operators are used for computational purposes during the typesetting phase, such as increasing counters, localizing strings like "theorem" and so on. A fundamental set of basic functional operators are built-in primitives. New functional operators can easily be added using the extern primitive. Functional operators operate on five main types of arguments: strings, numbers, lengths, booleans and tuples. Some operators are overloaded, so that they can be used for several types.

15.5.1. Operations on text

⟨length|*expr*⟩ (length of a string)

If *expr* is a string, the length of the string is returned. For instance, ⟨length|Hello⟩ evaluates to 5.

⟨range|*expr*|*start*|*end*⟩ (extract a substring)

Return the substring of *expr* starting at position *start* and ending at position *end* (not included). For instance, ⟨range|hottentottententententoonstelling|9|15⟩ evaluates to tenten.

⟨merge|*expr-1*|⋯|*expr-n*⟩ (concatenate strings)

This primitive may be used to concatenate several strings *expr-1* until *expr-n*. For instance, ⟨merge|Hello|World⟩ produces HelloWorld.

⟨number|*number*|*render-as*⟩ (alternative rendering of numbers)

Renders a *number* in a specified way. Supported values for *render-as* are

roman. Lower case Roman: ⟨number|18|roman⟩ ⟶ xviii.

Roman. Upper case Roman: ⟨number|18|Roman⟩ ⟶ XVIII.

alpha. Lower case letters: ⟨number|18|alpha⟩ ⟶ r.

Alpha. Upper case letters: ⟨number|18|Alpha⟩ ⟶ R.

⟨date⟩
⟨date|*format*⟩
⟨date|*format*|*language*⟩ (obtain the current date)

Returns the current date in a specified *format* (which defaults to a standard language-specific format when empty) and a specified *language* (which defaults to English). The format is similar to the one used by the UNIX `date` command. For instance, ⟨date⟩ evaluates to "April 27, 2005", ⟨date||french⟩ to "27 avril 2005" and ⟨date|%d %B om %k:%M|dutch⟩ to "27 April om 12:54".

⟨translate|*what*|*from*|*into*⟩ (translation of strings)

Returns the translation of a string *what* of the language *from* into the language *into*, using the built-in TeXmacs dictionaries. The languages should be specified in lowercase letters. For instance, ⟨translate|File|english|french⟩ yields "Fichier".

The list of currently available languages can be checked in the Document → Language menu. The built-in TeXmacs dictionaries can be found in

```
$TEXMACS_PATH/languages/natural/dic
```

When attempting to use a non-existing dictionary, the program may quit. For most purposes, it is more convenient to use the localize macro, which converts a string from English into the current language.

15.5.2. Arithmetic operations

⟨plus|*expr-1*|⋯|*expr-n*⟩

⟨minus|*expr-1*|⋯|*expr-n*⟩ (addition and subtraction)

Add or subtract numbers or lengths. For instance, ⟨plus|1|2.3|5⟩ yields 8.3 and ⟨plus|1cm|5mm⟩ produces ⟨tmlen|90708.6⟩. In the case of subtractions, the last argument is subtracted from the sum of the preceding arguments. For instance, ⟨minus|1⟩ produces -1 and ⟨minus|1|2|3|4⟩ yields 2.

⟨times|*expr-1*|⋯|*expr-n*⟩ (multiplication)

Multiply two numbers *expr-1* until *expr-n*. One of the arguments is also allowed to be a length, in which case a length is returned. For instance, ⟨times|3|3⟩ evaluates to 9 and ⟨times|3|2cm⟩ to ⟨tmlen|362835⟩.

⟨over|*expr-1*|⋯|*expr-n*⟩ (division)

Divide the product of all but the last argument by the last argument. For instance, ⟨over|1|2|3|4|5|6|7⟩ evaluates to 102.857, ⟨over|3spc|7⟩ to ⟨tmlen|2214.86|3320.14|4978.29⟩, and ⟨over|1cm|1pt⟩ to 28.4528.

⟨div|*expr-1*|*expr-2*⟩
⟨mod|*expr-1*|*expr-2*⟩ (division with remainder)

Compute the result of the division of an integer *expr-1* by an integer *expr-2*, or its remainder. For instance, ⟨div|18|7⟩=2 and ⟨mod|18|7⟩=4.

⟨equal|*expr-1*|*expr-2*⟩
⟨unequal|*expr-1*|*expr-2*⟩
⟨less|*expr-1*|*expr-2*⟩
⟨lesseq|*expr-1*|*expr-2*⟩
⟨greater|*expr-1*|*expr-2*⟩
⟨greatereq|*expr-1*|*expr-2*⟩ (comparing numbers or lengths)

Return the result of the comparison between two numbers or lengths. For instance, ⟨less|123|45⟩ yields false and ⟨less|123mm|45cm⟩ yields true.

15.5.3. Boolean operations

⟨or|*expr-1*|⋯|*expr-n*⟩
⟨and|*expr-1*|⋯|*expr-n*⟩

Returns the result of the boolean or/and on the expressions *expr-1* until *expr-n*. For instance, ⟨or|false|⟨equal|1|1⟩|false⟩ yields true.

⟨xor|*expr-1*|*expr-2*⟩

Returns the exclusive or of two expressions *expr-1* and *expr-2*, i.e. ⟨xor|true|true⟩ yields false.

⟨not|*expr*⟩

Returns the negation of *expr*.

15.5.4. Operations on tuples

⟨tuple|*expr-1*|⋯|*expr-n*⟩ (construct a tuple)

Forms a tuple from the expressions *expr-1* until *expr-n*.

⟨is-tuple|*expr*⟩ (tuple predicate)

Tests whether a given expression *expr* evaluates to a tuple.

⟨length|*expr*⟩ (length of a tuple)

If *expr* is a tuple, then we return its arity. For instance, ⟨length | ⟨tuple | hop | hola⟩⟩ evaluates to 2.

⟨look-up|*tuple*|*which*⟩ (access an entry in a tuple)

Returns the element with index *which* in *tuple*. For instance, ⟨look-up|⟨tuple|a|b|c⟩|1⟩ yields b.

⟨range|*expr*|*start*|*end*⟩ (extract a subtuple)

Return the subtuple of *expr* starting at position *start* and ending at position *end* (not included). For instance, ⟨range|⟨tuple|a|hola|hop|b|c⟩|2|4⟩ evaluates to ⟨tuple|hop|b⟩.

⟨merge|*expr-1*|⋯|*expr-n*⟩ (concatenate tuples)

This primitive may be used to concatenate several tuples *expr-1* until *expr-n*. For instance, ⟨merge|⟨tuple|1|2⟩|⟨tuple|3|4|5⟩⟩ produces ⟨tuple|1|2|3|4|5⟩.

15.6. TRANSIENT MARKUP

The tags described in this section are used to control the rendering of style files and style file elements. It both contains markup for activation and disactivation of content and for the rendering of tags.

⟨active|*content*⟩
⟨active*|*content*⟩
⟨inactive|*content*⟩
⟨inactive*|*content*⟩ (activation/disactivation of content)

These tags can be used to temporarily or permanently change the *activity* of the *content*. In usual documents, tags are by default active. In style files, they are by default inactive. For instance, an activated fraction is rendered as $\frac{1}{2}$; when deactivated, it is rendered as ⟨frac|1|2⟩.

The active and inactive tags only activate or desactivate the root tag of the *content*. Typically, a tag which contains hidden information (like hlink) can be disactivated by positioning the cursor just behind it and pressing `backspace`. This action just disactivates the hyperlink, but not the potentially complicated body of the hyperlink. Therefore, the hyperlink is transformed into an inactive tag of the form ⟨inactive|⟨hlink|*body*|*ref*⟩⟩.

The active* and inactive* variants are used to activate or disactivate the whole *content* (except when other (dis-)activation tags are found inside the *content*). The inactive* is used frequently inside the present documentation in order to show the inactive representation of TeXmacs content. Nevertheless, it is sometimes desirable to reactivate certain subtrees inside disactivated content. For instance, the following piece of disactivated code (using disactive*) contains the reactivated subexpression ♡♡♡ (using active*):

⟨assign|*love*|⟨macro|*from*|♡♡♡ from *from*.⟩⟩

⟨inline-tag|*name*|*arg-1*|⋯|*arg-n*⟩ (rendering of inline tags)

This tag is used for the default inline rendering of an inactive tag with a given *name* and arguments *arg-1* until *arg-n*. For instance, ⟨inline-tag|foo|x|y⟩ produces ⟨foo|x|y⟩. The style of the rendering may be customized in the Document → View → Source tags menu, or by modifying the *src-style*, *src-special*, *src-compact* and *src-close* environment variables.

⟨open-tag|*name*|*arg-1*|⋯|*arg-n*⟩
⟨middle-tag|*name*|*arg-1*|⋯|*arg-n*⟩
⟨close-tag|*name*|*arg-1*|⋯|*arg-n*⟩ (rendering of multi-line tags)

These tags are similar to inline-tag, when some of the arguments of the tag run over several lines. Typical HTML-like tags would correspond to ⟨open-tag|*name*⟩ and ⟨close-tag|*name*⟩. Since $\mathrm{T_EX_{MACS}}$ macros may take more than one argument, a middle-tag is provided for separating distinct multi-paragraph arguments. Moreover, the opening, middle and closing tags may take additional inline arguments for rendering in a compact fashion. For instance, the code

```
⟨open-tag|theorem⟩
⟨indent|The weather should be nice today.⟩
⟨close-tag|theorem⟩
```

is rendered by default as

```
⟨theorem|

    The weather should be nice today.

⟩
```

The rendering may be customized in a similar way as in the case of inline-tag.

⟨style-with|*var-1*|*val-1*|⋯|*var-n*|*val-n*|*body*⟩
⟨style-with*|*var-1*|*val-1*|⋯|*var-n*|*val-n*|*body*⟩ (alter presentation in style files only)

This tag may be used in order to temporarily modify the rendering of inactive tags, by setting each environment variable *var-i* to *val-i* in the local typesetting context of *body*. When importing a style file, each style-with/style-with* tag is replaced by its *body*. In the case of style-with, the modified rendering is only applied to the root tag of the *body*. In the case of style-with*, the rendering is modified for the entire *body*.

⟨style-only|⟨foo|*content*⟩⟩
⟨style-only*|⟨foo|*content*⟩⟩ (content for use in style files only)

This tag may be used in order to render an inactive tags as whether we applied the macro foo on it. When importing a style file, each style-only/style-only* tag is replaced by its *content*. In the case of style-only, the modified rendering is only applied to the root tag of the *content*. In the case of style-only*, the rendering is modified for the entire *content*.

⟨symbol|*symbol*⟩
⟨latex|*cmd*⟩
⟨hybrid|*cmd*⟩

⟨hybrid|*cmd*|*arg*⟩ (auxiliary tags for entering special content)

These tags are used only temporarily when entering special content.

When pressing C-q, a symbol tag is created. After entering the name of the symbol, or the ASCII-code of the symbol and pressing return, the symbol tag is replaced by the corresponding symbol (usually a string enclosed in <>).

When pressing \, a hybrid tag is created. After entering a string and pressing return, it is determined whether the string corresponds to a LaTeX command, a macro argument, a macro or an environment variable (in this order). If so, then the hybrid tag is replaced by the appropriate content. When pressing \ while a selection is active, then the selection automatically becomes the argument of the hybrid command (or the hybrid command itself, when recognized).

The latex tag behaves similarly as the hybrid tag except that it only recognizes LaTeX commands.

The rendering macros for source trees are built-in into TeXmacs. They should not really be considered as primitives, but they are not part of any style file either.

⟨indent|*body*⟩ (indent some content)

Typeset the *body* using some indentation.

⟨rightflush⟩ (indent some content)

Flush to the right. This macro is useful to make the end of a block environment run until the right margin. This allows for more natural cursor positioning and a better layout of the informative boxes.

⟨src-macro|*macro-name*⟩
⟨src-var|*variable-name*⟩
⟨src-arg|*argument-name*⟩
⟨src-tt|*verbatim-content*⟩
⟨src-integer|*interger*⟩
⟨src-length|*length*⟩
⟨src-error|*message*⟩ (syntactic highlighting on purpose)

These macros are used for the syntactic highlighting of source trees. They determine how to render subtrees which correspond to macro names, variable names, argument names, verbatim content, integers, lengths and error messages.

⟨src-title|*title*⟩
⟨src-style-file|*name*|*version*⟩
⟨src-package|*name*|*version*⟩
⟨src-package-dtd|*name*|*version*|*dtd*|*dtd-version*⟩ (style and package administration)

These macros are used for the identification of style files and packages and their corresponding D.T.D.s. The src-title is a container for src-style-file, src-package, src-package-dtd as well as src-license and src-copyright macros.

The src-style-file tag specifies the *name* and *version* of a style file and sets the environment variable with *name-style* to *version*. The src-package-dtd specifies the *name* and *version* of a package, as well as the corresponding *dtd* and its version *dtd-version*. It sets the environment variable *name-package* to *version* and *dtd-dtd* to *dtd-version*. The src-package tag is a shorthand for src-package-dtd when the name of the D.T.D. coincides with the name of the package.

15.7. Miscellaneous style-sheet primitives

⟨extern|*scheme-foo*|*arg-1*|⋯|*arg-n*⟩ (apply extern typesetting macro)

This primitive allows the user to implement macros in Scheme. The primitive applies the Scheme function or macro *scheme-foo* to the arguments *arg-1* until *arg-n*. For instance, the code ⟨extern|`(lambda (name) ‘(concat "hi " ,name))`|dude⟩ yields "hi dude".

The arguments *arg-1* until *arg-n* are evaluated and then passed as trees to *scheme-foo*. When defining a macro which relies on extern scheme code, it is therefore recommended to pass the macro arguments using the quote-arg primitive:

```
⟨assign|inc-div|
    ⟨macro|x|y|
        ⟨extern|
            (lambda (x y) ‘(frac ,x (concat "1+" ,y)))|
            ⟨quote-arg|x⟩|
            ⟨quote-arg|y⟩⟩⟩⟩
```

It has been foreseen that the accessability of the macro arguments x and y is preserved for this kind of definitions. However, since TeXmacs does not heuristically analyze your Scheme code, you will have to manually set the D.R.D. properties using drd-props.

Notice also that the Scheme function *scheme-foo* should only rely on secure scheme functions (and not on functions like `system` which may erase your hard disk). User implemented Scheme functions in plug-ins may be defined to be secure using the `:secure` option. Alternatively, the user may define all Scheme routines to be secure in Edit → Preferences → Security → Accept all scripts.

⟨write|*aux*|*content*⟩ (write auxiliary information)

Please document.

⟨flag|*content*|*color*⟩
⟨flag|*content*|*color*|*var*⟩ (display an informative flag)

This tag is used to in order to inform the user about information which is present in the document, but not visible when printed out. TeXmacs displays such informative flags for labels, formatting directives such as page breaks, and so on. In Document → View → Informative flags, the user may specify how the informative flags should be rendered.

The two-argument variant displays an informative flag with a given *content* and *color*. The *content* is only rendered when selecting Document → View → Informative flags → Detailed. For instance, ⟨flag|warning|red⟩ is rendered as . The optional *var* argument may be used in order to specify that the flag should only be visible if the macro argument *var* corresponds to an accessible part of the document. For instance, TeXmacs automatically generated labels for section titles (so as to include them in the table of contents), but it is undesirable to display informative flags for such labels.

15.8. Internal primitives

The primitives in this section are merely for internal use by TeXmacs only. They are documented for the sake of completeness, but you should only use them if you really know what you are doing.

⟨unknown⟩ (unknown content or unintialized data)

This primitive is mainly used for default uninialized value of environment variables; the main advantage of this tag is to be distinct from the empty string.

⟨error|*message*⟩ (error messages)

This primitive should never appear in documents. It is provided as aid in tracking down invalid constructs. It is produced at evaluation time by any kind of primitive which is given improper operands.

⟨collection|*binding-1*|···|*binding-n*⟩
⟨associate|*key*|*value*⟩ (collections of bindings)

The collection tag is used to represent hashtables with bindings *binding-1* until *binding-n*. Each binding is of the form ⟨associate|*key*|*value*⟩, with a *key* and an associated *value*.

⟨attr|*key-1*|*val-1*|···|*key-n*|*val-n*⟩ (XML-like attributes)

This tag is included for future compatability with XML. It is used for encoding XML-style attributes by $\mathrm{T_EX_{MACS}}$ trees. For instance, the fragment

```
<blah color="blue" emotion="verbose">
  Some XML stuff
</blah>
```

would typically be represented as

```
⟨blah|⟨attr|color|blue|emotion|verbose⟩|Some XML stuff⟩
```

⟨tag|*content*|*annotation*⟩
⟨meaning|*content*|*annotation*⟩ (associate a meaning to some content)

Associate a special meaning to some *content*. Currently, no real use has been made of these tags.

⟨backup|*save*|*stack*⟩ (save values on stack)

Used to represent temporarily saved values on a stack.

⟨dbox⟩ (marker for decorations)

This primitive is only intended for internal use by the datoms, dlines and dpages primitives.

⟨rewrite-inactive|*t*|*var*⟩ (internal primitive for rendering inactive markup)

This internal primitive is used for rewriting an inactive tree into a new tree whose rendering corresponds to the rendering of the inactive tree.

⟨new-dpage⟩
⟨new-dpage*⟩ (new double page)

Yet to be implemented primitives for starting a new double page.

⟨identity|*markup*⟩ (identity macro)

The identity macro is built-in into $\mathrm{T_EX_{MACS}}$. It should not really be considered as a primitive, but it is not part of any style file either.

In addition to these primitives for internal use only, there are also quite a few obsolete primitives, which are no longer being used by $\mathrm{T_EX_{MACS}}$, but whose names should be avoided when creating your own macros. The full list of obsolete primitives is: format, line-sep, with-limits, split, old-matrix, old-table, old-mosaic, old-mosaic-item, set, reset, expand, expand*, hide-expand, apply, begin, end, func, env, authorize.

Chapter 16

The standard TeXmacs styles

The user may select a major style from the Document → Style menu. The major style usually reflects the kind of document you want to produce (like a letter, an article or a book) or a particular layout policy (like publishing an article in a given journal). In addition to a major style, the user may select one or more additional packages from Document → Use package. Such packages may customize the major style, provide additional markup, or a combination of both.

In this chapter, we will survey the standard document styles and packages provided by TeXmacs. Most style files and packages have an abstract interface, the d.t.d. (data domain definition), which specifies which macros are exported by the style or package, and how to use them. Distinct styles or packages (like `header-article` and `header-book`) may share the same abstract interface, but differ in the way macros are rendered. For this reason, we will mainly be concerned with the description of the standard d.t.d.s, except when we focus on the rendering. Users may customize standard styles by defining new ones which match the same abstract interface (see the chapter on writing TeXmacs style files).

16.1. General organization

16.1.1. Standard TeXmacs styles

The main TeXmacs styles are:

`generic`. This is the default style when you open a new document. The purpose of this style is to produce quick, informal documents. For this reason, section numbering is disabled and the layout of paragraphs is very simple: instead of indenting the first lines of paragraphs, they are rather separated by white-space.

`article`. This style may be used for writing short scientific articles, which are subdivided into sections. The numbering of environments like theorems, remarks, etc. is relative to the entire document. If you use the `number-long-article` package, then the numbers are prefixed by the section number.

`book`. This is the basic style for writing books. Books are assumed to be subdivided into chapters and numbers of environments are prefixed by the chapter number. In general, it is also comfortable to store each chapter in a separate file, so that they can be edited more efficiently. This issue is explained in more detail in the section about books and multifile documents.

`seminar`. Documents based on this style are typically printed on slides for presentations using an overhead projector. You may also want to use it when making presentation directly from your laptop, after selecting View → Presentation mode. Notice however, that slides correspond to real pages, whereas you rather should use “switches” in presentation mode.

source. This is the privileged style for editing style files and packages. It enables "source mode", so that documents are rendered in a way which makes the structure fully apparent. For more details, we refer to the section on the rendering of style files.

The article style admits several variants, so as to make the layout correspond to the policy of specific journals. Currently, we have implemented the TeXmacs analogue of the LaTeX style amsart, as well as the styles acmconf and jsc. Similarly, we are developing styles tmarticle and tmbook which provide an alternative layout for articles and books.

In addition to variants of the article and book styles, TeXmacs provides also a few other styles, which are based on the main styles, but which provide some additional markup.

letter. This style is based on the informal generic style, but it provides additional markup for writing letters. The additional macro are mainly used for headers and endings of letters.

exam. This style, which is again based on generic, provides some additional markup for headers of exams. It also customizes the rendering of exercises.

tmdoc. This style is used for writing the TeXmacs documentation. It contains several tags for special types of content and extensions for linking, indexing, document traversal, etc.. Some aspects of this style are still under heavy development.

16.1.2. Standard TeXmacs packages

First of all, TeXmacs provides several packages for customizing the behaviour of the standard styles:

number-long-article. This package induces all numbers of environments (theorems, remarks, equations, figures, etc.) to be prefixed by the current section number. It is usually used in combination with the article style (for long articles) and the book style (for books with long chapters).

number-europe. By default, TeXmacs uses "American style numbering". This means that the same counter is used for numbering similar environments like theorem and proposition. In other words, a remark following "Theorem 3" will be numbered "Remark 4". If you want each environment to have its individual counter, then you should enable "European style numbering", by selecting the number-europe package.

number-us. This package may be used in order to switch back to American style numbering in the case when a third parties style file enforces European style numbering.

structured-list. This is an experimental package. By default, items in unnumbered lists or enumerations take no arguments and items in descriptions one argument. When using the structured-list package, they take an optional additional argument with the body of the item.

structured-section. This is an experimental package. By default, sectional tags only take a title argument. When using the structured-section package, they take an optional additional argument with the body of the section. Moreover, the environment rsection for recursive sections is provided.

varsession. This package may be used in order to obtain an alternative rendering of interactive sessions. The rendering is designed to be nice for interactive use, although less adequate for printing.

In addition to these packages, and the many packages for internal use, TeXmacs also provides a few personal example style packages allouche, bpr and vdh and several style packages for use in combination with external plug-ins (axiom, giac, macaulay2, etc.).

16.2. The common base for most styles

The std d.t.d. contains the markup which is common to virtually all styles. It is subdivided into the following parts:

16.2.1. Standard markup

Various standard markup is defined in std-markup. The following textual content tags all take one argument. Most can be found in the Text → Content tag menu.

⟨strong|*content*⟩

Indicates an **important** region of text. You can enter this tag via Text → Content tag → Strong.

⟨em|*content*⟩

Emphasizes a region of text like in "the *real* thing". This tag corresponds to the menu entry Text → Content tag → Emphasize.

⟨dfn|*content*⟩

For definitions like "a *gnu* is a horny beast". This tag corresponds to Text → Content tag → Definition.

⟨samp|*content*⟩

A sequence of literal characters like the ae ligature æ. You can get this tag via Text → Content tag → Sample.

⟨name|*content*⟩

The name of a particular thing or concept like the Linux system. This tag is obtained using Text → Content tag → Name.

⟨person|*content*⟩

The name of a person like Joris. This tag corresponds to Text → Content tag → Person.

⟨cite*|*content*⟩

A bibliographic citation like a book or magazine. Example: Melville's *Moby Dick*. This tag, which is obtained using Text → Content tag → Cite, should not be confused with cite. The latter tag is also used for citations, but where the argument refers to an entry in a database with bibliographic references.

⟨abbr|*content*⟩

An abbreviation. Example: I work at the C.N.R.S. An abbreviation is created using Text → Content tag → Abbreviation or the `A-a` keyboard shortcut.

⟨acronym|*content*⟩

An acronym is an abbreviation formed from the first letter of each word in a name or a phrase, such as HTML or IBM. In particular, the letters are not separated by dots. You may enter an acronym using Text → Content tag → Acronym.

⟨verbatim|*content*⟩

Verbatim text like output from a computer program. Example: the program said `hello`. You may enter verbatim text via Text → Content tag → Verbatim. The tag may also be used as an environment for multi-paragraph text.

⟨kbd|*content*⟩

Text which should be entered on a keyboard. Example: please type `return`. This tag corresponds to the menu entry Text → Content tag → Keyboard.

⟨code*|*content*⟩

Code of a computer program like in "`cout << 1+1;` yields 2". This is entered using Text → Content tag → Code. For longer pieces of code, you should use the code environment.

⟨var|*content*⟩

Variables in a computer program like in `cp` *`src-file dest-file`*. This tag corresponds to the menu entry Text → Content tag → Variable.

⟨math|*content*⟩

This is a tag which will be used in the future for mathematics inside regular text. Example: the formula $\sin^2 x + \cos^2 x = 1$ is well-known.

⟨op|*content*⟩

This is a tag which can be used inside mathematics for specifying that an operator should be considered on itself, without any arguments. Example: the operation $+$ is a function from $\mathbb{R}^2$ to $\mathbb{R}$. This tag may become depreciated.

⟨tt|*content*⟩

This is a physical tag for typewriter phase. It is used for compatibility with HTML, but we do not recommend its use.

Most of the following logical size tags can be found in Text → Size tag (or Mathematics → Size tag):

⟨really-tiny|*content*⟩, ⟨tiny|*content*⟩
⟨really-small|*content*⟩, ⟨very-small|*content*⟩, ⟨smaller|*content*⟩, ⟨small|*content*⟩
⟨normal-size|*content*⟩
⟨large|*content*⟩, ⟨larger|*content*⟩, ⟨very-large|*content*⟩, ⟨really-large|*content*⟩
⟨huge|*content*⟩, ⟨really-huge|*content*⟩

These logical size tags should be used by preference when typesetting parts of your document in a larger or smaller font. Environments like footnotes or captions of tables may also be based on logical size tags. Document styles from professional publishers often assign very precise font settings to each of the logical size tags. By default, the size tags are rendered as follows:

Really tiny
Tiny
Really small
Very small
Smaller
Small
Normal size
Large
Larger
Very large
Really large
Huge
Really huge

The following are standard environments:

⟨verbatim|*body*⟩

Described above.

⟨code|*body*⟩

Similar to code*, but for pieces of code of several lines.

⟨quote-env|*body*⟩

Environment for short (one paragraph) quotations.

⟨quotation|*body*⟩

Environment for long (multi-paragraph) quotations.

⟨verse|*body*⟩

Environment for poetry.

⟨center|*body*⟩

This is a physical tag for centering one or several lines of text. It is used for compatibility with HTML, but we do not recommend its use.

Some standard tabular environments are

⟨tabular*|*table*⟩

Centered tables.

⟨block|*table*⟩

Left aligned tables with a border of standard `1ln` width.

⟨block*|*table*⟩

Centered tables with a border of standard `1ln` width.

The following miscellaneous tags don't take arguments:

⟨TeXmacs⟩

The TeXmacs logo.

⟨TeXmacs-version⟩

The current version of TeXmacs (1.0.5).

⟨made-by-TeXmacs⟩

A macro which may be used to indicate that your document was written using TeXmacs.

⟨TeX⟩

The TeX logo.

⟨LaTeX⟩

The LaTeX logo.

⟨hrule⟩

A horizontal rule like the one you see below:

The following miscellaneous tags all take one or more arguments:

⟨phantom|*content*⟩

This tag takes as much space as the typeset argument *content* would take, but *content* is not displayed. For instance, ⟨phantom|phantom⟩ yields "   ".

⟨overline|*content*⟩

For $\overline{\text{overlined text}}$, which can be wrapped across several lines.

⟨underline|*content*⟩

For underlined text, which can be wrapped across several lines.

⟨fold|*summary*|*body*⟩

The *summary* is displayed and the *body* ignored: the macro corresponds to the folded presentation of a piece of content associated to a short title or abstract. The second argument can be made visible using Insert → Switch → Unfold.

⟨unfold|*summary*|*body*⟩

Unfolded presentation of a piece of content *body* associated to a short title or abstract *summary*. The second argument can be made invisible using Insert → Switch → Fold.

⟨switch|*current*|*alternatives*⟩

Content which admits a finite number of alternative representation among which the user can switch using the function keys F9, F10, F11 and F12. This may for instance be used in interactive presentations. The argument *current* correspond to the currently visible presentation and *alternative* to the set of alternatives.

16.2.2. Standard symbols

The `std-symbol` d.t.d. defines the special symbols ¢, ¤, ¥, ©, ⓐ, ®, °, ², ³, ¹, µ, ¶, ¼, ½, ¾, € and ™. It also provides the macro nbsp for non-breakable spaces.

As soon as the font support will be further improved, this d.t.d. should become obsolete.

16.2.3. Standard mathematical markup

Standard mathematical markup is defined in `std-math`.

⟨binom|*among*|*nr*⟩

For binomial coefficients, like $\binom{n}{m}$.

⟨choose|*among*|*nr*⟩

Alternative name for binom, but depreciated.

⟨shrink-inline|*among*|*nr*⟩

A macro which switches to scriptsize text when you are not in display style. This macro is mainly used by developers. For instance, the binom macro uses it.

The following are standard mathematical tabular environments:

⟨matrix|*table*⟩

For matrices $M = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$.

⟨det|*table*⟩

For determinants $\Delta = \begin{vmatrix} 1 & 2 \\ 3 & 4 \end{vmatrix}$.

⟨choice|*table*⟩

For choice lists $|x| = \begin{cases} -x, & \text{if } x \leqslant 0 \\ x, & \text{if } x \geqslant 0 \end{cases}$.

16.2.4. Standard lists

16.2.4.1. Using list environments

The standard TeXmacs lists are defined in `std-list`. The unnumbered lists environments are:

⟨itemize|*body*⟩

The tag before each item depends on the nesting depth.

⟨itemize-minus|*body*⟩

Uses $-$ for the tag.

⟨itemize-dot|*body*⟩

Uses $\bullet$ for the tag.

⟨itemize-arrow|*body*⟩

Uses $\rightarrow$ for the tag.

The following environments can be used for producing numbered lists:

⟨enumerate|*body*⟩

The kind of number before each item depends on the nesting depth.

⟨enumerate-numeric|*body*⟩

Number the items by 1, 2, 3, etc.

⟨enumerate-roman|*body*⟩

Number the items by i, ii, iii, etc.

⟨enumerate-Roman|*body*⟩

Number the items by I, II, III, etc.

⟨enumerate-alpha|*body*⟩

Number the items by a), b), c), etc.

⟨enumerate-Alpha|*body*⟩

Number the items by A), B), C), etc.

The following environments can be used for descriptive lists:

⟨description|*body*⟩

The environment for default descriptive lists (usually description-compact).

⟨description-compact|*body*⟩

Align the left hand sides of the items in the list and put their descriptions shortly behind it.

⟨description-dash|*body*⟩

Similar to description-compact, but use a — to separate each item from its description.

⟨description-align|*body*⟩

Align the left hand sides of the descriptions, while aligning the items to the right.

⟨description-long|*body*⟩

Put the items and their descriptions on distinct lines.

New items in a list are indicated through the item tag or the item* tag in the case of descriptions. The item tag takes no arguments and the item* tag one argument. When using the experimental `structured-list` package, these tags may take an optional body argument. In the future, all list items should become structured.

By default, items in sublists are numbered in the same way as usual lists. Each list environment *list* admits a variant *list** whose items are prefixed by the last item in the parent list. Of course, this feature can be used recursively.

16.2.4.2. Customization of list environments

The `std-list` provides the following redefinable macros for customizing the rendering of lists and items in lists:

⟨render-list|*body*⟩

This block environment is used to render the *body* of the list. Usually, the macro indents the body and puts some vertical space around it.

⟨aligned-item|*item-text*⟩

This inline macro is used to render the *item-text* in a right-aligned way. As a consequence, text after such items will appear in a left-aligned way.

⟨compact-item|*item-text*⟩

This inline macro is used to render the *item-text* in a left-aligned way. As a consequence, text after such items may be indented by the width of the *item-text* (except when the text is rendered on a different paragraph).

16.2.5. Automatic content generation

The `std-automatic` d.t.d. contains macros for the automatic generation and rendering of auxiliary content. There are four main types of such content in TeXmacs: bibliographies, tables of contents, indexes and glossaries. Other types of automatically generated content like lists of figures are usually similar to one of the four above types (in the case of lists of figures, we use glossaries). The rendering of the entire sections which contain the bibliographies, tables of contents, etc. are specified in the `section-base` d.t.d..

16.2.5.1. Bibliographies

The following macros may be used in the main text for citations to entries in a bibliographic database.

⟨cite|*ref-1*|···|*ref-n*⟩

Each argument *ref-i* is a citation corresponding to an item in a BiB-TeX file. The citations are displayed in the same way as they are referenced in the bibliography and they also provide hyperlinks to the corresponding references. The citations are displayed as question marks if you did not generate the bibliography.

⟨nocite|*ref-1*|···|*ref-n*⟩

Similar as cite, but the citations are not displayed in the main text.

⟨cite-detail|*ref*|*info*⟩

A bibliographic reference *ref* like above, but with some additional information *info*, like a chapter or a page number.

The following macros may be redefined if you want to customize the rendering of citations or entries in the generated bibliography:

⟨render-cite|*ref*⟩

Macro for rendering a citation *ref* at the place where the citation is made using cite. The *content* may be a single reference, like "TM98", or a list of references, like "Euler1, Gauss2".

⟨render-cite-detail|*ref*|*info*⟩

Similar to render-cite, but for detailed citations made with cite-detail.

⟨render-bibitem|*content*⟩
⟨transform-bibitem|*content*⟩

At the moment, bibliographies are generated by BibTEX and imported into TEXMACS. The produced bibliography is a list of bibliographic items with are based on special LATEX-specific macros (bibitem, block, protect, etc.). These macros are all defined internally in TEXMACS and eventually boil down to calls of the render-bibitem, which behaves in a similar way as item*, and which may be redefined by the user.

The transform-bibitem is used to "decorate" the *content*. For instance, transform-bibitem may put angular brackets and a space around *content*. Notice that the standard implementation of render-bibitem macro is based on transform-bibitem.

⟨bib-list|*largest*|*body*⟩

The individual "bibitems" are enclosed in a bib-list, which behaves in a similar way as the description environment, except that we provide an extra parameter *largest* which contains a good indication about the largest width of an item in the list.

16.2.5.2. Tables of contents

The following macros may be used in the main text for adding entries to the table of contents. They are automatically called by most sectional macros, but it is sometimes desirable to manually add additional entries.

⟨toc-main-1|*entry*⟩
⟨toc-main-2|*entry*⟩

Create an important *entry* in the table of contents. The macro toc-main-1 is intended to be used only for very important entries, such as parts of a book; it usually has to be added manually. The macro toc-main-2 is intended to be used for chapter or sections. Important entries are usually displayed in a strong font.

⟨toc-normal-1|*entry*⟩
⟨toc-normal-2|*entry*⟩
⟨toc-normal-3|*entry*⟩

Add a normal *entry* to the table of contents, of different levels of importance. Usually, toc-normal-1 corresponds to sections, toc-normal-2 to subsections and toc-normal-3 to subsubsections.

⟨toc-small-1|*entry*⟩
⟨toc-small-2|*entry*⟩

Add an unimportant *entry* to the table of contents, like a paragraph. Since such entries are not very important, some styles may simply ignore the toc-small-1 and toc-small-2 tags.

By redefining the following macros, it is possible to customize the rendering of tables of contents:

⟨toc-strong-1|*content*|*where*⟩
⟨toc-strong-2|*content*|*where*⟩

Used for rendering table of contents entries created using toc-main-1 resp. toc-main-2.

⟨toc-1|*content*|*where*⟩
⟨toc-2|*content*|*where*⟩

⟨toc-3|*content*|*where*⟩
⟨toc-4|*content*|*where*⟩
⟨toc-5|*content*|*where*⟩

Used for rendering table of contents entries created using toc-normal-1, toc-normal-2, toc-normal-3, toc-small-1 resp. toc-small-2.

⟨toc-dots⟩

The separation between an entry in the table of contents and the corresponding page number. By default, we use horizontal dots.

16.2.5.3. Indexes

The following macros may be used in the main text for inserting entries into the index.

⟨index|*primary*⟩

Insert *primary* as a primary entry in the index.

⟨subindex|*primary*|*secondary*⟩

Insert *secondary* in the index as a subentry of *primary*.

⟨subsubindex|*primary*|*secondary*|*ternary*⟩

Similar to subindex but for subsubentries *ternary*.

⟨index-complex|*key*|*how*|*range*|*entry*⟩

Insert complex entries into the index. This feature is documented in detail in the section about index generation.

⟨index-line|*key*|*entry*⟩

Adds *entry* to the index, by sorting it according to *key*.

The following macros may be redefined if you want to customize the rendering of the index:

⟨index-1|*entry*|*where*⟩
⟨index-2|*entry*|*where*⟩
⟨index-3|*entry*|*where*⟩
⟨index-4|*entry*|*where*⟩
⟨index-5|*entry*|*where*⟩

Macro for rendering an *entry* in the index on page(s) *where*. The macro index-1 corresponds to principal entries, the macro index-2 to secondary entries, and so on.

⟨index-1*|*entry*⟩
⟨index-2*|*entry*⟩
⟨index-3*|*entry*⟩
⟨index-4*|*entry*⟩
⟨index-5*|*entry*⟩

Similar to index-1 until index-5, but without the page number(s).

⟨index-dots⟩

Macro for producing the dots between an index entry and the corresponding page number(s).

16.2.5.4. Glossaries

The following macros may be used in the main text for inserting glossary entries.

⟨glossary|*entry*⟩

Insert *entry* into the glossary.

⟨glossary-dup|*entry*⟩

For creating an additional page number for an *entry* which was already inserted before.

⟨glossary-explain|*entry*|*explanation*⟩

A function for inserting a glossary *entry* with its *explanation*.

⟨glossary-line|*entry*⟩

Insert a glossary *entry* without a page number.

The following macros can be redefined if you want to customize the rendering of the glossary:

⟨glossary-1|*entry*|*where*⟩

Macro for rendering a glossary entry and its corresponding page number(s).

⟨glossary-2|*entry*|*explanation*|*where*⟩

Macro for rendering a glossary entry, its explanation, and its page number.

⟨glossary-dots⟩

Macro for producing the dots between a glossary entry and the corresponding page number(s).

16.2.6. Utilities for writing style files

The `std-utils` package provides several macros which may be useful when writing style files. First of all, the following macros may be used for rendering purposes:

⟨hflush⟩
⟨left-flush⟩
⟨right-flush⟩

Low level tags for flushing to the right in the definition of environments. One usually should use wide-normal or wide-centered instead.

⟨wide-normal|*body*⟩
⟨wide-centered|*body*⟩

These tags are used to make the *body* span over the entire paragraph width. The text is left-aligned in the case of wide-normal and centered in the case of wide-centered. Making a body span over the entire paragraph width does not change the rendering on paper, but it facilitates the editing on the document. Indeed, on the one hand side, the box which indicates that you are inside the environment will span over the entire paragraph width. On the other hand, when clicking sufficiently close to the text inside this box, it becomes easier to position your cursor at the start or at the end inside the environment. You may check this by clicking on one of the texts below:

>Some text inside a wide-normal environment. <

> Some text inside a wide-centered environment. <

⟨padded-normal|*space-above*|*space-below*|*body*⟩
⟨padded-centered|*space-above*|*space-below*|*body*⟩

These tags are variants of ⟨wide-normal|*body*⟩ and ⟨wide-centered|*body*⟩, which put some vertical white space *space-above* and *space-below* above and below the *body*.

⟨wide-bothlined|*top-border*|*bot-border*|*top-sep*|*bot-sep*|*body*⟩
⟨wide-std-bothlined|*body*⟩
⟨padded-bothlined|*space-above*|*space-below*|*top-border*|*bot-border*|*top-sep*|*bot-sep*|*body*⟩
⟨padded-std-bothlined|*space-above*|*space-below*|*body*⟩
⟨wide-underlined|*bborder*|*bsep*|*body*⟩
⟨wide-std-underlined|*body*⟩

These tags are used to make the *body* span over the entire paragraph width and to put a horizontal rule above and/or below it. The widths of the rules are given by *top-border* and *bot-border* and the separation between the rules by *top-sep* and *bot-sep*. The standard width and separation (used by wide-std-bothlined, padded-std-bothlined and wide-std-underlined) are `1ln` and `1sep`. The padded variants specify additional spaces *space-above* and *space-below* above and below the rules. As an example, ⟨wide-std-underlined|left⟨htab|5mm⟩right⟩ yields:

left right

Wide underlined environments are typically used for page headers. Wide environments which are both overlined and underlined are typically used for abstracts or floating figures and tables.

⟨wide-framed|*border-width*|*hsep*|*vsep*|*body*⟩
⟨wide-std-framed|*body*⟩
⟨wide-framed-colored|*border-color*|*body-color*|*border-width*|*hsep*|*vsep*|*body*⟩
⟨wide-std-framed-colored|*border-color*|*body-color*|*body*⟩

These tags put the *body* inside a frame box which spans over the whole paragraph. The user may specify a *border-width*, horizontal and vertical separations *hsep* and *vsep* between the border and the text, and colors *border-color* and *body-color* for the border and the background. For instance, ⟨wide-std-framed-colored|brown|pastel green|Hi there!⟩ yields

Hi there!

⟨indent-left|*left-amount*|*body*⟩
⟨indent-right|*right-amount*|*body*⟩
⟨indent-both|*left-amount*|*right-amount*|*body*⟩

These environments may be used in order to increase the current left and/or right indentation by the amounts *left-amount* and/or *right-amount*.

⟨margin-first-other|*first-margin*|*other-margin*|*body*⟩

This environment allows to set the margin *first-margin* for the first lines of paragraphs in the *body*, as well as the margin *other-margin* for the other lines. This environment is for instance useful for glossaries, indexes, etc., in which case *other-margin* is often larger than *first-margin*. Notice that this environment enables indentation for the first line of *body*.

The following macros may be used in order to set headers and footers:

⟨set-header|*header-text*⟩

A macro for permanently changing the header. Notice that certain tags in the style file, like sectional tags, may override such manual changes.

⟨set-footer|*footer-text*⟩

A macro for permanently changing the footer. Again, certain tags in the style file may override such manual changes.

⟨blanc-page⟩

Remove all headers and footers from this page.

⟨simple-page⟩

Remove the header of this page and set the footer to the current page number (centered). This macro is often called for title pages or at the start of new chapters.

Other macros provided by `std-utils` are:

⟨localize|*text*⟩

This macro should be used in order to "localize" some English text to the current language. For instance, ⟨with|*language*|french|⟨localize|Theorem⟩⟩ yields Théorème.

⟨map|*fun*|*tuple*⟩

This macro applies the macro *fun* to each of the entries in a *tuple* (or the children of an arbitrary TeXmacs tag) and returns the result as a tuple. For instance, ⟨map|⟨macro|x|⟨em|x⟩⟩|⟨tuple|1|2|3⟩⟩ yields ⟨quote|⟨tuple|*1*|*2*|*3*⟩⟩ (the quote only appears when rendering the result, not when performing further computations with it).

16.2.7. Counters and counter groups

In TeXmacs, all automatic numbering of theorems, sections, etc. is done using "counters". Such counters may be individual counters (like *equation-nr*) or belong to a group of similar counters (like in the case of *theorem-nr*). TeXmacs allows for the customization of counters on an individual or groupwise basis. Typically, you may redefine the rendering of a counter (and let it appear as roman numerals, for instance), or undertake special action when increasing the counter (such as resetting a subcounter).

New individual counters are defined using the following meta-macro:

⟨new-counter|x⟩

Defines a new counter with name x. The counter is stored in the numerical environment variable *x-nr* and in addition, the following macros are defined:

⟨the-*x*⟩

Retrieve the counter such as it should be displayed on the screen.

⟨reset-*x*⟩

Reset the counter to 0.

⟨inc-*x*⟩

Increase the counter. This macro may also be customized by the user so as to reset other counters (even though this is not the way things are done in the standard style files).

⟨next-*x*⟩

Increase the counter, display the counter and set the current label.

For the purpose of customization, the new-counter macro also defines the following macros:

⟨display-*x*|nr⟩

This is the macro which is used for transforming the numerical value of the counter into the value which is displayed on the screen.

⟨counter-*x*|x⟩

This internal macro is used in order to retrieve the name of the environment variable which contains the counter. By default, this macro returns "nr-x", but it may be redefined if the counter belongs to a group.

As noticed in the introduction, $\mathrm{T_EX_{MACS}}$ uses *counter groups* in order to make it possible to treat similar counters in a uniform way. For instance the counter group `theorem-env` regroups the counters `theorem`, `proposition`, `lemma`, etc.. New counter groups and are defined using:

⟨new-counter-group|g⟩

Create a new counter group with name g. This results in the creation of the following macros:

⟨display-in-*g*|x|nr⟩
⟨counter-in-*g*|x⟩

These macros are similar to the macros display-*x* and counter-*x* from above, but relative to the counter group. The name x of the counter in consideration is passed as an argument.

New counters can be added to the group using:

⟨add-to-counter-group|x|g⟩

Defines a new counter x and add it to the counter group g. For counters in groups, the macros display-*x* and counter-*x* are replaced with the corresponding macros display-in-*g* and counter-in-*g* for their groups. Nevertheless, two new macros ind-display-*x* and ind-counter-*x* are defined which may take over the roles of display-*x* and counter-*x* in the case when the group consists of individual counters.

At any moment, you may decide whether the counters of a group share a common group counter, or whether they all use their individual counters. This feature is used for instance in order to switch between American style numbering and European style numbering:

⟨group-common-counter|g⟩

Use a common counter for the group (which is stored in the environment variable g-nr).

⟨group-individual-counters|g⟩

Use an individual counter for each member of the group (this is the default).

We notice that group counters may recursively belong to super-groups. For instance, the following declarations are from `env-base.ts`:

⟨new-counter-group|std-env⟩
⟨new-counter-group|theorem-env⟩
⟨add-to-counter-group|theorem-env|std-env⟩
⟨group-common-counter|theorem-env⟩

16.2.8. Special markup for programs

The program d.t.d. provides markup for the layout of computer programs. However, these tags should be considered as very unstable, since we plan to replace them by a set of more detailed tags:

⟨algorithm|*name*|*body*⟩

The *name* of the algorithm and its *body*, which includes its possible specification.

⟨body|*body*⟩

The real body of the algorithm.

⟨indent|*content*⟩

For indenting part of an algorithm.

16.2.9. Special markup for sessions

The session d.t.d. provides the following environments for computer algebra sessions:

⟨session|*body*⟩

Environment for marking a session. All macros below are only for use inside sessions.

⟨input|*prompt*|*body*⟩

An input field with a *prompt* and the actual input.

⟨output|*body*⟩

An output field.

⟨textput|*body*⟩

Fields with ordinary text. These may for instance be used for comments and explanations.

⟨errput|*body*⟩

This macro is used inside output fields for displaying error messages.

In fact, these environments are based on environments of the form *lan*-session, *lan*-input, *lan*-output, *lan*-textput and *lan*-errput for every individual language *lan*.

If language-specific environments do not exist, then generic-session, generic-input, generic-output, generic-textput and generic-errput are taken instead. It is recommended to base the language-specific environments on the generic ones, which may have different implementations according to the style (e.g. the varsession package). For this purpose, we also provide the generic-output* environment, which is similar to generic-output, except that margins remain unaltered.

16.3. Standard environments

The `env` d.t.d. contains the standard environments which are available in most styles. It is subdivided into the following parts:

16.3.1. Defining new environments

The `env-base` d.t.d. contains high-level markup which can be used by the user to define new numbered environments for theorems, remarks, exercises and figures:

⟨new-theorem|*env-name*|*display-name*⟩

This meta-macro is used for defining new theorem-like environments. The first argument *env-name* specifies the name for the environment (like "experiment") and *display-name* the corresponding text (like "Experiment"). When defining a new theorem-like environment like experiment, an unnumbered variant experiment* is automatically defined as well.

⟨new-remark|*env-name*|*display-name*⟩

Similar as new-theorem, but for remarks.

⟨new-exercise|*env-name*|*display-name*⟩

Similar as new-theorem, but for exercises.

⟨new-exercise|*env-name*|*display-name*⟩

Similar as new-theorem, but for figures. When defining a new type of figure, like "picture", the new-figure macro defines both the inline environment small-picture and the block-environment big-picture, as well as the unnumbered variants small-picture* and big-picture*.

The theorem-like and remark-like environments belong to a common counter-group `theorem-env`. By default, we use American-style numbering (one common counter for all environments). When selecting the package `number-europe`, each environment uses its own counter. All exercises and figures use their own counter-group.

More generally, the `std-env` counter-group regroups the counters for all standard TeXmacs environments. Typically, all counters in this group are prefixed in a similar way (for instance by the number of the chapter). Figure 16.1 shows how the hierarchical organization of this counter group.

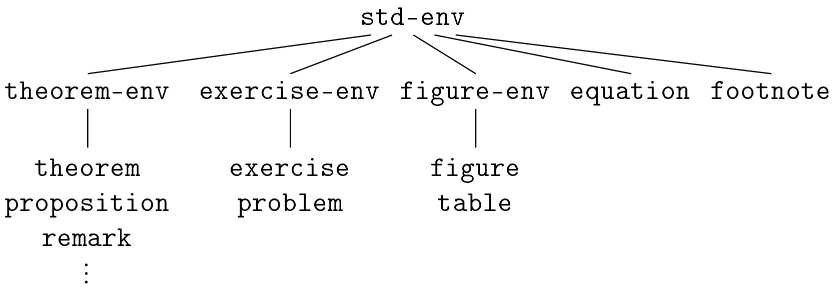


Figure 16.1. Organization of the counters for the standard TeXmacs environments.

In addition to the standard theorem-like, remark-like, exercise-like and figure-like environments, other numbered textual environments may be defined using the new-env macro. These environments may be based on arbitrary counter-groups:

⟨new-env|*group*|*env*|*env-name*|*display-name*⟩

The first argument is the name of the counter *group* to which the new environment belongs. The second argument *env* is the name of a binary macro for rendering the environment. The arguments of the rendering macro are a name (like "Theorem 3.14") and its body. The remaining arguments are similar as for new-theorem. For instance, in the standard style-sheets, new-theorem is defined by

> ⟨assign|*new-theorem*|⟨macro|*env*|*name*|⟨new-env|*env*|*name*|theorem-env|render-theorem⟩⟩⟩

We recall that you may add new counters or counter-groups to the `theorem-env` counter-group using the new-counter-group and add-to-counter-group macros, as described in the section about counters.

16.3.2. Mathematical environments

The `env-math` d.t.d. specifies which mathematical environments can be used inside text-mode. In other words, the environments should be used inside text-mode, but their bodies contain mathematical formulas or tables of mathematical formulas.

⟨equation|*body*⟩

A numbered equation.

⟨equation*|*body*⟩

An unnumbered equation.

⟨eqnarray|*table*⟩

An array of numbered equations (not yet implemented).

⟨eqnarray*|*table*⟩

An array of unnumbered equations.

Inside the eqnarray* environment, you can use the eq-number tag in order to number the equation.

WARNING 16.1. The numbering of equations inside tables is not yet as it should be. In particular, the eqnarray tag is equivalent to eqnarray* at the moment. Later on, when the eqnarray tag will be implemented correctly, you will also have a no-number tag in order to suppress the number of an equation, and a style package for numbering equations at the left hand side.

WARNING 16.2. There is no option for numbering equations at the left hand side available yet. Nevertheless, you may use the manual tag leq-number for this. You also have a tag next-number which directly display the next number and increases the equation counter.

Warning 16.3. We do not encourage the use of the AMS-TeX environments `align`, `gather` and `split`. Nevertheless, they are available under the names align, gather, eqsplit together with their variants align*, gather* and eqsplit*. In the future, we plan to provide more powerful environments.

16.3.3. Theorem-like environments

16.3.3.1. Using the theorem-like environments

The `env-theorem` d.t.d. contains the default theorem-like and other textual environments, which are available through Text → Environment. They are subdivided into three main categories:

Variants of theorems. The bodies of theorem-like environments are usually emphasized. By default, the following such environments are available via Text → Environment: theorem, proposition, lemma, corollary, axiom, definition, notation, conjecture.

Variants of remarks. The following ones are available via Text → Environment: remark, example, note, warning, convention.

Variants of exercises. Two such environments are provided by default and available via Text → Environment: exercise and problem.

The environments are all available in unnumbered versions theorem*, proposition*, etc. as well. You may use `A-*` in order to switch between the unnumbered and numbered version. The following tags are also provided:

⟨proof|*body*⟩

For proofs of theorems.

⟨dueto|*who*⟩

An environment which can be used to specify the inventors of a theorem. It should be used at the start inside the body of a theorem, like in

Theorem. (Pythagoras) $a^2 + b^2 = c^2$.

16.3.3.2. Customization of the theorem-like environments

The following customizable macros are used for the rendering of textual environments:

⟨render-theorem|*name*|*body*⟩

This macro is used for displaying a theorem-like environments. The first argument *name* specifies the name of the theorem, like "Theorem 1.2" and the second argument *body* contains the body of the theorem. This environment is used for environments defined by new-theorem.

⟨render-remark|*name*|*body*⟩

Similar to render-theorem, but for remark-like environments.

⟨render-exercise|*name*|*body*⟩

Similar to render-theorem, but for exercise-like environments.

⟨render-proof|*name*|*body*⟩

Similar to render-theorem, but for proofs. This environment is mainly used for customizing the name of a proof, like in “End of the proof of theorem 1.2”.

Notice that you may also use these macros if you want an environment which is rendered in a similar way as a theorem, but with another name (like “Corollary of Theorem 7”).

The following tags can be used for further customization of the rendering:

⟨theorem-name|*name*⟩

This macro controls the appearance of the names of theorem-like *and* remark-like environments. Most styles use bold face or small capitals.

⟨exercise-name|*name*⟩

Similar to theorem-name, but for exercises.

⟨theorem-sep⟩

The separator between the name of a theorem-like or remark-like environment and its main body. By default, this is a period followed by a space.

⟨exercise-sep⟩

Similar to theorem-sep, but for exercises.

Each standard environment *x* also comes with a customizable macro *x*-text which renders the localized name of the environment. For instance, ⟨with|*language*|dutch|⟨theorem-text⟩⟩ yields “Stelling”.

16.3.4. Environments for floating objects

16.3.4.1. Using the environments for floating objects

The `env-float` d.t.d. provides the following environments for floating objects:

⟨small-figure|*body*|*caption*⟩

This macro produces an inline figure with *body* as its main body and *caption* as a caption. Inline figures may for instance be used to typeset several small figures side by side inside a floating object.

⟨big-figure|*body*|*caption*⟩

This macro produces a big figure with *body* as its main body and *caption* as a caption. Big figures span over the whole paragraph width.

⟨small-table|*body*|*caption*⟩

Similar to small-figure, but for tables.

⟨big-table|*body*|*caption*⟩

Similar to big-figure, but for tables.

⟨footnote|*body*⟩

Produces a footnote.

The figure-like environments also admit unnumbered versions small-figure*, big-figure*, etc., which are obtained using A-*.

16.3.4.2. Customization of the environments for floating objects

The following macros can be used for customizing the rendering of figure-like environments:

⟨render-small-figure|*aux*|*name*|*body*|*caption*⟩

This macro is used for rendering small figure-like environments. The first argument *aux* specifies an auxiliary channel (like "figure" or "table") which is used for inserting the caption inside the list of figures. The second argument *name* specifies the name of the figure (like "Figure 2.3" or "Table 5"). The last arguments *body* and *caption* correspond to the figure itself and a caption.

⟨render-big-figure|*aux*|*name*|*body*|*caption*⟩

Similar to render-small-figure, but for displaying a big figure-like environments.

The following tags can be used for customizing the appearance the text around figures, tables and footnotes:

⟨figure-name|*name*⟩

This macro controls the appearance of the text "Figure". By default, we use bold face.

⟨figure-sep⟩

This macro produces the separator between the figure and its number and the caption. By default, it produces a period followed by a space.

⟨footnote-sep⟩

This macro produces the separator between the number of the footnote and the text. By default, it produces a period followed by a space.

16.4. HEADERS AND FOOTERS

16.4.1. Standard titles

16.4.1.1. Entering titles and abstracts

The `header-title` d.t.d. provides tags for entering information about the entire document. The two top-level tags are

⟨doc-data|*data-1*|···|*data-n*⟩

Specify data attached to your document (title, authors, etc.; see below) and render the title.

⟨abstract|*body*⟩

The abstract for your paper.

When creating a doc-data tag using Text → Title → Insert title, TeXmacs automatically inserts a doc-title tag as its first arguments. New data may be inserted from the Text → Title menu. Each child *data-1*, ..., *data-n* of the doc-data tag is of one of the following forms:

⟨doc-title|*title*⟩

Specify the *title* of the document.

⟨doc-subtitle|*subtitle*⟩

Specify the *subtitle* of the document.

⟨doc-author-data|*data-1*|···|*data-n*⟩

Specify the data for one of the authors of the document (name, address, etc.; see below).

⟨doc-date|*date*⟩

The creation date of the document. In particular you may take ⟨date⟩ for the value of *date* for the current date.

⟨doc-running-title|*title*⟩

Specify a running *title* for your document which may be used in page headers.

⟨doc-running-author|*author*⟩

Specify a running *author* for your document which may be used in page headers.

⟨doc-keywords|*kw-1*|···|*kw-n*⟩

Specify keywords *kw-1* until *kw-n* for your document.

⟨doc-AMS-class|*nr-1*|···|*nr-n*⟩

Specify A.M.S. subject classification numbers *nr-1* until *nr-n* for your document.

⟨doc-note|*note*⟩

A note about your document. In particular, you may take ⟨with-TeXmacs-text⟩ for the value of *note* in order to indicate that your document has been written using TeXmacs.

When inserting an additional author using Text → Title → Author → Insert author, TeXmacs inserts a doc-author-data tag with an author-name tag as its first argument. New author data may be inserted from the Text → Title → Author menu. Each child *data-1*, ..., *data-n* of the doc-author-data tag is of one of the following forms:

⟨author-name|*name*⟩

Specify the *name* of the author.

⟨author-address|*address*⟩

An *address* where the author can be reached.

⟨author-email|*email*⟩

An *email* address for the author.

⟨author-homepage|*homepage*⟩

The *homepage* of the author.

⟨author-note|*note*⟩

A *note* attached to the author, like a thank-word.

As a general rule, the use of any of the subtags of doc-data or doc-author-data is optional. An individual subtag may also be specified several times. This is useful for documents with several authors, or authors with several addresses. The rendering of title information is very style-dependent: some styles render addresses in a single line or even as a footnote, where other styles use a more widely spaced presentation. Often, some information like keywords or AMS subject classification numbers are only rendered as a part of the abstract.

16.4.1.2. Customizing the global rendering of titles

Depending on the kind of attributes, complex titles often use several rendering styles in a simultaneous version. More precisely, a title usually consists of the following parts:

- A well visible part at the top of the title page.
- Additional notes, which are displayed in the footer.
- An potentially invisible part, with information like running titles and authors.
- A postponed part, which is only rendered in the abstract.

Similarly, individual authors may also contain a main part, which is rendered as part of the title, and an additional part, which is rendered as a footnote. Moreover, the layout often changes if the paper has more than one author.

The $\mathrm{T_EX_{MACS}}$ mechanism for rendering titles therefore relies on several macros which extract the information corresponding to each of the above parts. This process may also involve some sorting, like putting the authors before the date or *vice versa*. At a second stage, each extracted part of the title is passed to the appropriate rendering macro. The following macros are used for extracting title information:

⟨doc-data-main|*data-1*|⋯|*data-n*⟩
⟨doc-data-main*|*data-1*|⋯|*data-n*⟩

> This macro only keeps and sorts the data which should be displayed in the main title. The doc-data-main* variant is used in the case when the document has more than one author.

⟨doc-data-note|*data-1*|⋯|*data-n*⟩

> This macro only keeps and sorts the data which should be displayed as a footnote.

⟨doc-data-abstract|*data-1*|⋯|*data-n*⟩

> This macro only keeps and sorts the data which should be displayed in the abstract.

⟨doc-data-hidden|*data-1*|⋯|*data-n*⟩

> This macro only keeps and sorts the data which might or should not be displayed at all.

In a similar fashion, the following macros are used for extracting author information:

⟨doc-author-main|⟨doc-author-data|*data-1*|⋯|*data-n*⟩⟩

> This macro only keeps and sorts the data which should be displayed inside the main title.

⟨doc-author-note|*data-1*|⋯|*data-n*⟩

> This macro only keeps and sorts the data which should be displayed as a footnote.

It should be noticed that each of the above macros should return a document tag with the selected data as its children. For instance,

```
⟨doc-author-main|
    ⟨author-address|Somewhere in Africa⟩|
    ⟨author-name|The big GNU⟩|
    ⟨author-note|Very hairy indeed!⟩⟩
```

should typically return

```
⟨document|
    ⟨author-address|Somewhere in Africa⟩|
    ⟨author-name|The big GNU⟩⟩
```

The only exception to this rule is doc-data-hidden which should return a concat tag instead.

16.4.1.3. Customizing the rendering of title fields

Both title information and author information is rendered as a vertical stack of “title blocks” and “author blocks”. The following macros may be used to customize the global rendering of such blocks:

⟨doc-title-block|*content*⟩
⟨doc-author-block|*content*⟩

Macros for rendering one component of the title or author information.

The following macros may be used to customize the rendering of title information; notice that they are usually built on top of doc-title-block.

⟨doc-make-title|*content*⟩

This macro is used for the rendering of the main title information. Usually, it contains at least the title itself, as well as one or several authors.

⟨doc-render-title|*title*⟩

This macro is used for rendering the *title* of the document. The doc-title macro also takes care of rendering references to potential footnotes.

⟨doc-subtitle|*title*⟩

This macro is used for rendering the *subtitle* of the document.

⟨doc-author|*content*⟩

In the case when the document has a single author, then this macro is used for rendering the *content* information about him or her.

⟨doc-authors|*content*⟩

In the case when the document has several authors, then this macros is used for rendering all author-related *content* which is part of the main title.

⟨doc-date|*date*⟩

This macro is used for rendering the creation *date* of the document.

The following macros may be used to customize the rendering of author information; notice that they are usually built on top of doc-author-block.

⟨author-render-name|*name*⟩

Renders the *name* of the author.The author-name macro also takes care of rendering references to potential footnotes.

⟨author-by|*name*⟩

A macro which may put the text "by " in front of the *name* of an author.

⟨author-address|*address*⟩

Renders the *address* of the author.

⟨author-email|*email*⟩

Renders the *email* address of the author.

⟨author-homepage|*email*⟩

Renders the *homepage* of the author.

The following macros are used for information which is usually not rendered as a part of the main title, but rather as a footnote or part of the abstract.

⟨doc-title-note|*note*⟩
⟨doc-author-note|*note*⟩

A macro for rendering a *note* attached to the document or one of its authors. The note will usually appear as part of a footnote. By default, notes that consist of several lines are compressed into a single paragraph.

⟨doc-keywords|*kw-1*|⋯|*kw-n*⟩

A macro for displaying a list of keywords.

⟨doc-AMS-class|*nr-1*|⋯|*nr-n*⟩

A macro for displaying a list of A.M.S. subject classification numbers.

16.4.2. Standard headers

The `header` d.t.d. provides call-back macros which allow page headers and footers to change automatically when specifying the title information of the document or when starting a new section.

⟨header-title|*title*⟩

This macro is called when specifying the *title* of a document.

⟨header-author|*author*⟩

This macro is called when specifying the *author*(s) of a document.

⟨header-primary|*section-title*|*section-nr*|*section-type*⟩

This macro is called at the start of each new primary section (e.g. chapter for book style, or section for article style). The *section-type* is a literal text like "Chapter" or "Section".

⟨header-secondary|*section-title*|*section-nr*|*section-type*⟩

This macro is called at the start of each new secondary section (e.g. section for book style, or subsection for article style). The *section-type* is a literal text like "Section" or "Paragraph".

In style files, page headers and footers are usually set by the above call-back macros, and not manually. You may directly modify headers and footers by setting the corresponding environment variables or using several helper macros supplied by `std-utils`.

16.5. LATEX STYLE SECTIONS

16.5.1. Using sectional tags

The `section-base` d.t.d. provides the standard tags for sections, which are the same as in LATEX. Most sectional tags take one argument: the name of the section. The intention of the following tags is to produce numbered sections:

⟨part|*title*⟩
⟨chapter|*title*⟩
⟨section|*title*⟩
⟨subsection|*title*⟩
⟨subsubsection|*title*⟩
⟨paragraph|*title*⟩
⟨subparagraph|*title*⟩
⟨appendix|*title*⟩

The intention of this macro is to produce a numbered title for a part (resp. chapter, section, subsection, etc.). The numbering is not required, but merely an intention: the paragraph and subparagraph tags are usually not numbered and some styles (like the generic style) do not produce numbers at all.

The tags part*, chapter*, section*, subsection*, subsubsection*, paragraph*, subparagraph* and appendix* can be used for producing the unnumbered variants of the above tags.

By default, all sectional only produce the section title. When using the experimental package `structured-section`, all sectional tags are enriched, so that they take the body of the section as an optional argument. Moreover, an additional tag rsection is provided in order to produce recursively embedded sections. For instance, an rsection inside a section behaves like a subsection. In the future, all list items should become structured.

The `section-base` d.t.d. also provides the following sectional environments with automatically generated content

⟨bibliography|*aux*|*style*|*file-name*|*body*⟩

This macro is used for producing bibliographies. The first argument *aux* specifies the auxiliary channel with the data for generating the bibliography (`bib`, by default). The arguments *style* and *file-name* contain the bibliography style and the file with the bibliographic database. The *body* argument corresponds to the automatically generated content.

⟨table-of-contents|*aux*|*body*⟩

This macro is used for producing tables of contents. The first argument *aux* specifies the auxiliary channel with the data for generating the bibliography (`toc`, by default). The *body* argument corresponds to the automatically generated content.

⟨the-index|*aux*|*body*⟩

Similar to table-of-contents but for indices and default channel `idx`.

⟨the-glossary|*aux*|*body*⟩
⟨list-of-figures|*aux*|*body*⟩
⟨list-of-tables|*aux*|*body*⟩

Similar to table-of-contents but for glossaries (default channel `gly`), lists of figures (default channel `figure`) and lists of tables (default channel `table`).

The above tags also admit the variants bibliography*, table-of-contents*, the-index* and the-glossary* with an additional argument *name* before *body*, which specifies the name of the section. For instance, the the-glossary* was formerly used for lists of figures and lists of tables.

16.5.2. Customization of the sectional tags

The `section-base` d.t.d. also contains many tags for customizing the rendering of sections and other section-related behaviour. The following two tags affect all sections:

⟨sectional-sep⟩

A macro for customizing the separator between the number of a section and its title. By default, we use two spaces.

⟨sectional-short-style⟩

A predicate which tells whether documents for this style are intended to be short or long. When sectional-short-style evaluates to `true`, then appendices, bibliographies, etc. are supposed to be special types of sections. Otherwise, they will be special types of chapters.

For each sectional tag *x*, the following tags are provided for customization:

⟨*x*-text⟩

A macro which displays the (localized) name of the sectional environment. For instance, ⟨with|*language*|french|⟨appendix-text⟩⟩ produces "Annexe".

⟨*x*-title|*title*⟩

A macro for displaying the unnumbered section title.

⟨*x*-numbered-title|*title*⟩

A macro for displaying the numbered section title.

⟨*x*-display-numbers⟩

A predicate which specifies whether numbers will really be displayed. For instance, in the case of paragraph, this macro evaluates to false. Consequently, even though *x*-numbered-title *does* display the paragraph number, the main macro *x* will call *x*-title and not *x*-numbered-title, so that paragraph titles are not numbered.

⟨*x*-sep⟩

A macro for customizing the separator between the number of a section and its title. By default, we call sectional-sep.

⟨x-clean⟩

A hook for resetting all subcounters of the section.

⟨x-header|*name*⟩

A hook for changing the page headers.

⟨x-toc|*name*⟩

A hook for putting the section title into the table of contents.

Finally, the `section-base` d.t.d. provides rendering macros render-table-of-contents, render-bibliography, render-index and render-glossary, each of which takes two arguments: the name of the section and its body. It also provides the macros prologue-text, epilogue-text, bibliography-text, table-of-contents-text, index-text, glossary-text, list-of-figures-text and list-of-tables-text for customizing the names of special sections.

16.5.3. Helper macros for rendering section titles

The `section-base` d.t.d. contains several helper macros which can (should) be used when customizing the rendering of section titles:

⟨sectional-short|*body*⟩
⟨sectional-short-italic|*body*⟩
⟨sectional-short-bold|*body*⟩

These macros should be used for rendering "short section titles", for which the section body starts immediately at the right of the title. Usually, titles of paragraphs and subparagraphs are rendered in a short fashion, while the other section titles span over the entire width of a paragraph.

⟨sectional-normal|*body*⟩
⟨sectional-normal-italic|*body*⟩
⟨sectional-short-bold|*body*⟩

These macros should be used for rendering "normal left-aligned section titles". Such titles span over the entire paragraph width.

⟨sectional-centered|*body*⟩
⟨sectional-centered-italic|*body*⟩
⟨sectional-centered-bold|*body*⟩

These macros should be used for rendering "normal centered section titles". Such titles span over the entire paragraph width.

Chapter 17

Compatibility with other formats

TeXmacs is fully compatible with Postscript (as well as PDF), which is used as the format in order to print documents. TeXmacs also provides converters from and to LaTeX and an input filter for Html.

17.1. Compatibility with LaTeX

Although TeXmacs has not been designed to be fully compatible with LaTeX, it is possible to convert documents from TeXmacs to LaTeX and *vice versa*, although the result will not always be perfect. Also, conversions from TeXmacs to LaTeX will generally yield better results than conversions the other way around. In particular, TeXmacs may reasonably well be used to write articles, which need to be converted to LaTeX for submission purposes. In this chapter, we will describe more precisely the conversion mechanisms, which will help you to obtain a result as satisfactory as possible.

17.1.1. Conversion from TeXmacs to LaTeX

The most common situation is that you want to convert an article from TeXmacs to LaTeX, in order to submit it to some journal. Given a TeXmacs file `name.tm`, you may convert it into a LaTeX file `name.tex` using File → Export → Latex. At a first stage, you may try to run LaTeX on `name.tex`, and see whether you obtain a satisfactory result. If so, then you should submit `name.tex` together with the style file `TeXmacs.sty`, which can be found in the directory `$TEXMACS_PATH/misc/latex`.

Often, the journal to which you submit uses its own style file, say `journal.sty`. In that case, you should also copy the file

```
$TEXMACS_PATH/styles/article.ts
```

to

```
~/.TeXmacs/styles/journal.ts
```

and use `journal` as your document style in Document → Style → Other. You may optionally edit `journal.ts`, so that the article layout becomes closer to the journal's style. In some cases, you also have to create a new copy of `TeXmacs.sty`, and modify some of the environments for compatibility with the journal's style file `journal.sty`.

If your first try to convert your document into LaTeX did not yield a satisfactory result, then you will usually observe that only minor parts of the texts were not converted correctly. This may be due to three main reasons:

- Your text uses some specific TeXmacs features.
- You used a TeXmacs feature, which has not yet been implemented in the conversion algorithm.

- You found a bug in the conversion algorithm.

These issues will be discussed in more detail in the next section.

In case of problems, a naive strategy would be to correct the produced LaTeX file and to send it to the journal. However, this strategy has the disadvantage that you have to make these corrections over and over again, each time that you convert your TeXmacs file `name.tm`, after having made some extra modifications. A better strategy is to use the Insert → Specific → Latex and Insert → Specific → Texmacs constructs to write text which is visible in the converted resp. original file only.

For instance, assume that the word "blauwbilgorgel" is hyphenated correctly in the TeXmacs source, but not in the LaTeX conversion. Then you may proceed as follows:

1. Select "blauwbilgorgel".
2. Click on Insert → Specific → Texmacs to make the text "blauwbilgorgel" TeXmacs-specific.
3. Click on Insert → Specific → Latex.
4. Type the latex code `blauw\-bil\-gor\-gel` with the correct hyphenation.
5. Press `return` to activate the LaTeX-specific text.

In a similar fashion, you may insert LaTeX-specific line breaks, page breaks, vertical space, style parameter modifications, etc.

17.1.2. Possible conversion problems

17.1.2.1. Specific TeXmacs features

Some TeXmacs typesetting primitives have no analogues in LaTeX, and the conversion algorithm will simply transform them into blank space. Some main features which are specific to TeXmacs are the following:

- Left primes.
- Big separators between big parentheses.
- Mosaics.
- Trees.
- Complex user macros.
- Vertical spaces "before" and "after".
- Indentation flags "before" and "after".

You should avoid to use these specific TeXmacs features, if your document needs to be converted into LaTeX. Nevertheless, in the far future, the conversion program might generate encapsulated postscript by default of a more intelligible translation.

17.1.2.2. Not yet implemented conversions

Although we try to keep the conversion algorithm as complete as possible for your needs, certain things have not yet been implemented. Some examples of not yet implemented issues are

- Non standard fonts.
- Conversion of tabulars.
- Style parameters.

Any suggestion about desirable extensions of the conversion algorithm should be reported to

`contact@texmacs.org`

and we will try to incorporate it as quickly as possible. It may take some time to implement the correct conversion of style parameters, since these are not the same in TeXmacs and LaTeX. Furthermore, layout differences between TeXmacs and LaTeX can not entirely be eliminated.

17.1.2.3. Bugs in the conversion algorithm

The most annoying situation if when a converted TeXmacs document produces lots of errors at the compilation or if the result has nothing to do with the original. In that case you have probably detected a bug in the conversion algorithm (or in the installation of LaTeX on your system). Please try to figure out the source of the bug in this case and report it by sending an email to

`TeXmacs@math.u-psud.fr`

17.1.2.4. Work-arounds

TeXmacs has not been designed to be fully compatible with LaTeX. As to the conversion from LaTeX to TeXmacs, our main aim is to *help* you in converting old documents to TeXmacs. As long as you did not define weird environments and as long as you did not use weird style files and commands, you should be able to convert your old documents reasonably well. Otherwise, we suggest to modify your old document in a way that is does convert reasonably well and to apply some final changes in the result.

17.1.3. Conversion from LaTeX to TeXmacs

The current aim of the conversion program from LaTeX to TeXmacs, is to *help* you in translating old documents into TeXmacs. *Grosso modo*, conversions from LaTeX to TeXmacs are more problematic than conversions the other way around. Nevertheless, as long as you restrict yourself to using the most common LaTeX commands, you should be able to convert your old documents reasonably well. For example, all TeXmacs help files have been written in LaTeX in order to validate the LaTeX to TeXmacs conversion program.

You may convert a LaTeX document `name.tex` into TeXmacs using File → Import → Latex and save it under `name.tm`. If your LaTeX document was written sufficiently well, then the converted result should be more or less acceptable, apart from certain unrecognized commands, which appear in red. A good solution would be to write your own style file for converted documents, based on the original style, and in which the unrecognized commands are defined.

However, in certain less fortunate cases, the converted document will look like a great mess. This usually stems from the fact that TEX and LATEX allow users to modify the parser dynamically, for instance using the `\catcode` command. In this case, the conversion program may get confused, by making erroneous assumptions on the mode or the environment. As a result, text may be converted as mathematics, mathematics as verbatim, and so on. Nevertheless, the commands in your source file `name.tex` which confused the conversion program are usually easily localized by comparing the LATEX source with its TEXMACS conversion. Modulo some hacking of the source, you should be able to remove the litigious code, so that the document converts reasonably well.

In the future, we also plan to extend the conversion program with a style file converter and some additional features which facilitate the translation of user defined commands, which are defined in another document than the one you want to convert.

17.2. Conversion of TEXMACS documents to Html

We have started to implemented the conversion between HTML and TEXMACS. At this moment, it is only possible to import HTML documents using File → Import → Html. Most of HTML 2.0 and parts of HTML 3.0 are currently supported. However, no browsing facilities have been added yet. In the future, we plan to implement Math-ML.

When importing HTML documents, files whose names start with `http:` or `ftp:` will be downloaded from the web using `wget`. If you compiled TEXMACS yourself, then you can download `wget` from

```
ftp://ftp.gnu.org/pub/gnu/wget/
```

In the binary distributions, we have included `wget`.

17.3. Adding new data formats and converters

Using the GUILE/SCHEME extension language, it is possible to add new data formats and converters to TEXMACS in a modular way. Usually, the additional formats and converters are declared in your personal `~/.TeXmacs/progs/my-init-texmacs.scm` or a dedicated plug-in. Some examples may be found in the directory `$TEXMACS_PATH/progs/convert`, like init-html.scm.

Declaring new formats.

A new format is declared using the command

```
(define-format format
  (:name format-name)
  options)
```

Here ***format*** is a symbol which stands for the format and ***format-name*** a string which can be used in menus. In fact, a data format usually comes in several variants: a format ***format***`-file` for files, a format ***format***`-document` for entire documents, a format ***format***`-snippet` for snippets, like selections, and ***format***`-object` for the preferred internal scheme representation for doing conversions (i.e. the parsed variant of the format). Converters from ***format***`-file` to ***format***`-document` and *vice versa* are provided automatically.

The user may specify additional options for the automatic recognition of formats by their file suffix and contents. The possible suffixes for a format, with the default one listed first, may be specified using

```
(:suffix default-suffix other-suffix-1 ... other-suffix-n)
```

A (heuristic) routine for recognizing whether a given document matches the format can be specified using either one of the following:

```
(:recognize predicate)
(:must-recognize predicate)
```

In the first case, suffix recognition takes precedence over document recognition and in the second case, the heuristic recognition is entirely determined by the document recognition predicate.

Declaring new converters.

New converters are declared using

```
(converter from to
  options)
```

The actual converter is specified using either one of the following options:

```
(:function converter)
(:function-with-options converter-with-options)
(:shell prog prog-pre-args from progs-infix-args to prog-
post-args)
```

In the first case, the ***converter*** is a routine which takes an object of the ***from*** format and returns a routine of the ***to*** format. In the second case, the ***converter*** takes an additional association list as its second argument with options for the converter. In the last case, a shell command is specified in order to convert between two file formats. The converter is activated only then, when ***prog*** is indeed found in the path. Also, auxiliary files may be created and destroyed automatically.

TeXmacs automatically computes the transitive closure of all converters using a shortest path algorithm. In other words, if you have a converter from x to y and a converter from y to z, then you will automatically have a converter from x to z. A "distance between two formats via a given converter" may be specified using

```
(:penalty floating-point-distance)
```

Further options for converters are:

```
(:require cond)
(:option option default-value)
```

The first option specifies a condition which must be satisfied for this converter to be used. This option should be specified as the first or second option and always after the `:penalty` option. The `:option` option specifies an option for the converter with its default value. This option automatically become a user preference and it will be passed to all converters with options.

Appendix A
Configuration of TeXmacs

A.1. Introduction

Before you start using TeXmacs, it may be wise to configure the program first in Edit → Preferences, so that it will fit your needs best. Most importantly, you should choose a "look and feel" in Edit → Preferences → Look and feel. This will enable you for instance to let the keyboard shortcuts used by TeXmacs be similar to what you are used to in other applications.

Also, TeXmacs comes with a powerful keyboard shortcut system, which attempts to optimize the use of the modifier keys like `shift` and `control` on your keyboard. However, on certain systems these modifier keys are not well configured, so that you may wish to redo this yourself.

A.2. Configuration of the modifier keys

TeXmacs uses five major keyboard modifiers: `shift`, `control`, `alternate`, `meta` and `hyper`, which are abbreviated as `S-`, `C-`, `A-`, `M-` and `H-`. The `shift` and `control` keys are present on virtually all keyboards and the `alternate` key on almost all. Most keyboards for PC's nowadays also have a `windows` key, which is usually equivalent to `meta` for TeXmacs.

Before reconfiguring your keyboard, you should first check that this is indeed necessary. If you have keys which correspond to `shift`, `control`, `alternate` and `meta` in a suitable way, then you probably do not want to do anything. A possible exception is when you want to use a simple key like `caps-lock` for typing mathematical symbols. In that case, you should map `caps-lock` to `hyper`.

In order to reconfigure the keyboard, you simply select the logicial modifier that you want to correspond to a given physical key in Edit → Preferences → Keyboard. For instance, selecting Windows key → Map to M modifier, the `windows` key will correspond to the `meta` modifier. Similarly, when selecting Caps-lock key → Map to H modifier, the `caps-lock` key will correspond to the `hyper` modifier.

Unfortunately, X Window only allows system-wide reconfiguration. Consequently, if you reconfigure the `caps-lock` key inside TeXmacs, then the new behaviour of `caps-lock` will affect all other applications too. It is therefore important to reconfigure only those keys which you do not use for something else in other applications. For instance, the `windows` key is not used by many applications, so it generally does not do any harm to reconfigure it. You may also prefer to perform an appropriate system-wide configuration. This can be done using the `xmodmap` command; see the corresponding manual page for more information.

In certain cases, you already have keys on your keyboard which correspond to `alternate`, `meta` and `hyper`, but not in the way you want. This can be done by remapping the `A-`, `M-` and `H-` prefixes to other logical modifiers in the first group of submenus of Edit → Preferences → Keyboard.

For instance, for Emacs compatability, you might want to permute the `meta` or `windows` key with `alternate` without making any system-wide changes. This can be done by finding out which modifiers correspond to these keys; usually this will be `Mod1` for `alternate` and `Mod4` for `meta` or `windows`. We next perform the necessary permutation in Edit → Preferences → Keyboard, by selecting A modifier → Equivalent for Mod4 and M modifier → Equivalent for Mod1.

A.3. Notes for Russian and Ukranian users

In order to type Russian (and similarly for Ukranian) text, you may several options:

- Select Russian as your default language in Edit → Preferences → Language → Russian. If TeXmacs starts with Russian menus, then this is done automatically if the Russian locale is set.
- Select Russian for an entire document using Document → Language → Russian.
- Select Russian for a portion of text in another document using Format → Language → Russian.

If your X server uses the xkb extension, and is instructed to switch between the Latin and Russian keyboard modes, you need not do anything special. Just switch your keyboard to the Russian mode, and go ahead. All the software needed for this is included in modern Linux distributions, and the xkb extension is enabled by default in `XF86Config`. With the xkb extension, keysyms are 2-byte, and Russian letters are at 0x6??. The keyboard is configured by `setxkbmap`. When X starts, it issues this command with the system-wide `Xkbmap` file (usually living in `/etc/X11/xinit`), if it exists; and then with the user's `~/.Xkbmap`, if it exists. A typical `~/.Xkbmap` may look like

```
ru basic grp:shift_toggle
```

This means that the keyboard mode is toggled by `l-shift r-shift`. Other popular choices are `control shift` or `control alternate`, see `/usr/X11R6/lib/X11/xkb/` for more details. This is the preferred keyboard setup for modern Linux systems, if you plan to use Russian often.

In older Linux systems, the xkb extension is often disabled. Keysyms are 1-byte, and are configured by `xmodmap`. When X starts, it issues this command with the system-wide `Xmodmap` (usually living in `/etc/X11/xinit`), if it exists; and then with the user's `~/.Xmodmap`, if it exists. You can configure the mode toggling key combination, and use a 1-byte Russian encoding (such as koi8-r) in the Russian mode. It is easier to download the package `xruskb`, and just run

```
xrus jcuken-koi8
```

at the beginning of your X session. This sets the layout jcuken (see below) and the encoding koi8-r for your keyboard in the Russian mode. If you use such keyboard setup, you should select Options → international keyboard → russian → koi8-r.

It is also possible to use the Windows cp1251 encoding instead of koi8-r, though this is rarely done in UNIX. If you do use `xrus jcuken-cp1251`, select cp1251 instead of koi8-r.

All the methods described above require some special actions to "russify" the keyboard. This is not difficult, see the Cyrillic-HOWTO or, better, its updated version `http://www.inp.nsk.su/~baldin/Cyrillic-HOWTO-russian/Cyrillic-HOWTO-russian.html`

Also, all of the above methods globally affect all X applications: text editors (emacs, nedit, kedit...), xterms, TeXmacs etc.

If you need to type Russian only once, or very rarely, a proper keyboard setup may be more trouble than it's worth. For the benefit of such occasional users, TeXmacs has methods of Russian input which require no preliminary work. Naturally, such methods affect only TeXmacs, and no other application.

The simplest way to type some Russian on the standard US-style keyboard with no software setup is to select Edit → Preferences → Keyboard → Cyrillic input method → translit. Then, typing a Latin letter will produce "the most similar" Russian one. In order to get some Russian letters, you have to type 2- or 3-letter combinations:

Shorthand	for	Shorthand(s)	for
`A-" e`	ё	`A-" E`	Ё
`y o`	ё	`Y o` `Y O`	Ё
`z h`	ж	`Z h` `Z H`	Ж
`j tab`	ж	`J tab`	Ж
`c h`	ч	`C h` `C H`	Ч
`s h`	ш	`S h` `S H`	Ш
`s c h`	щ	`S c h` `S C H`	Щ
`e tab`	э	`E tab`	Э
`y u`	ю	`Y u` `Y U`	Ю
`y a`	я	`Y a` `Y A`	Я

Table A.1. Typing Cyrillic text on a Roman keyboard.

If you want to get, e.g., "сх", and not "ш", you have to type `s / h`. Of course, the choice of "optimal" mapping of Latin letters to Russian ones in not unique. You can investigate the mapping supplied with TeXmacs and, if you don't like something, override it in your `~/.TeXmacs/progs/my-init-texmacs.scm`.

If you select jcuken instead of translit, you get the "official" Russian typewriter layout. It is so called because the keys "qwerty" produce "йцукен". This input method is most useful when you have a Russian-made keyboard, which has additional Russian letters written on the key caps in red, in the jcuken layout (a similar effect can be achieved by attaching transparent stickers with red Russian letters to caps of a US-style keyboard). It is also useful if you are an experienced Russian typist, and your fingers remember this layout.

Those who have no Russian letters indicated at the key caps often prefer the yawerty layout, where the keys "qwerty" produce "яверты". Each Latin letter is mapped into a "similar" Russian one; some additional Russian letters are produced by `shift`-digits. TeXmacs comes with a slightly modified yawerty layout, because it does not redefine the keys `$`, `£`, `\`, which are important for TeXmacs, are not redefined. The corresponding Russian letters are produced by some `shift`-digit combinations instead.

Appendix B

About GNU TeXmacs-1.0.5

B.1. Summary

GNU TeXmacs	
Installed version	1.0.5
Supported systems	Most GNU/Linux systems
Copyright	© 1998–2002 by Joris van der Hoeven
License	GNU General Public License
Web sites	`http://www.texmacs.org` `http://www.gnu.org/software/texmacs`
Contact	`contact@texmacs.org`
Regular mail	Dr. Joris van der Hoeven Dépt. de Mathématiques (Bât. 425) Université Paris-Sud 91405 Orsay Cedex France

Table B.1. Summary of the principal information about GNU TeXmacs.

B.2. The philosophy behind TeXmacs

B.2.1. A short description of GNU TeXmacs

GNU TeXmacs is a free scientific text editor, which was both inspired by TeX and GNU Emacs. The editor allows you to write structured documents via a wysiwyg (what-you-see-is-what-you-get) and user friendly interface. New styles may be created by the user. The program implements high-quality typesetting algorithms and TeX fonts, which help you to produce professionally looking documents.

The high typesetting quality still goes through for automatically generated formulas, which makes TeXmacs suitable as an interface for computer algebra systems. TeXmacs also supports the Guile/Scheme extension language, so that you may customize the interface and write your own extensions to the editor.

Converters exist for TeX/LaTeX and they are under development for Html/MathML/Xml. In the future, TeXmacs is planned to evolve towards a complete scientific office suite, with spreadsheet capacities, a technical drawing editor and a presentation mode.

GNU TeXmacs is hosted by the Centre de Ressources Informatiques de Haute Savoie, Archamps, France.

B.2.2. Why freedom is important for scientists

One major objective of TeXmacs is to promote the development of free software for and by scientists, by significantly reducing the cost of producing high quality user interfaces. If you plan to write an interface between TeXmacs and other software, then please contact us.

As a mathematician, I am deeply convinced that only free programs are acceptable from a scientific point of view. I see two main reasons for this:

- A result computed by a “mathematical” system, whose source code is not public, can not be accepted as part of a mathematical proof.
- Just as a mathematician should be able to build theorems on top of other theorems, it should be possible to freely modify and release algorithms of mathematical software.

However, it is strange, and a shame, that the main mathematical programs which are currently being used are proprietary. The main reason for this is that mathematicians often do not consider programming as a full scientific activity. Consequently, the development of useful software is delegated to “engineers” and the resulting programs are used as black boxes.

This subdivision of scientific activity is very artificial: it is often very important from a scientific point of view to know what there is in the black box. Inversely, deep scientific understanding usually leads to the production of better software. Consequently, I think that scientists should advocate the development of software as a full scientific activity, comparable to writing articles. Then it is clear too that such software should be diffused in a way which is compatible with the requirements of science: public availability, reproducibility and free usability.

B.3. The authors of TeXmacs

The GNU TeXmacs system, which is part of the GNU project, was designed and written by Joris van der Hoeven. The system was inspired both by the TeX system, written by D. Knuth, and by Emacs, written by R. Stallman. Special thanks goes to them, as well as to the C.N.R.S. (the French national institute for scientific research), which employs me and authorized me to freely distribute this program. Further thanks go to the contributors below.

B.3.1. Developers of TeXmacs

- Dan Martens made the Windows port.
- Andrey Grozin has constantly helped us with many issues: interfaces to several computer algebra systems, support for Cyrillic, tools for the manipulation of dictionaries, etc.
- David Allouche replaced the gencc preprocessor by the more standard C++ template system. He also made many other patches, bug reports and he did a lot of the administration of TeXmacs.
- Henri Lesourd is working on native technical drawing support in TeXmacs. He also fixed a bug in the presentation mode.

- Andreas Seidl has been helping with documentation, a Cygwin package and several other things.
- Dan Grayson helped me to implement communications with computer algebra systems via pipes. He also provided some money support for TeXmacs, and he made many useful comments and suggestions.
- Karim Belabas designed and developed with me the first protocol for interfacing TeXmacs with scientific computation or computer algebra systems. He also implemented the interface with the Pari system.
- Felix Breuer helped with the support of Unicode and other character encodings. He also made a donation to the project.
- Josef Weidendorfer made several patches for improving the performance of TeXmacs.
- Stéphane Payrard made an important bugfix for destroying windows.
- Josef Weidendorfer for two patches to improve the speed of TeXmacs.
- Johann Dréo for the new TeXmacs icon and many other graphics.
- Bill Page and David Mentré for the support of the free version of Axiom.
- Chu-Ching Huang for writing CAS documentation and making a Knoppix CD for TeXmacs.
- Mickael Floc'hlay and Arnaud Ébalard for their work on searching for help.
- Gwenael Gabard for some fixes in the LaTeX to TeXmacs converter.
- Igor V. Kovalenko and Teemu Ikonen for their help on debugging TeXmacs and a few patches.
- Gareth McCaughan made several patches and comments.
- Immanuel Normann is working on an OpenMath converter.
- Jonas Lööf for a precise installation procedure on Cygwin.
- Rob Clark made a patch which improves the system time support.
- Stanislav Brabec for several patches so as to increase portability.

B.3.2. Authors and maintainers of plugins for TeXmacs

Axiom — Andrey Grozin, Bill Page, David Mentré and Tim Daly.

DraTeX — Nicolas Ratier.

Eikleides — Mark Arrasmith.

Giac — Bernard Parisse.

GNUplot — Stephan Mucha.

Graphviz — Jorik Blaas.

GTybalt — Stefan Weinzierl.

Macaulay 2 — Dan Grayson.

Maple — Christian Even.

Mathemagix — Joris van der Hoeven.

Maxima — Andrey Grozin and James Amundson.

Mupad — Christopher Creutzig and Andrey Grozin.

Octave — Michael Graffam.

Pari — Karim Belabas.

Python — Ero Carrera.

Qcl — Andrey Grozin.

R — Michael Lachmann.

Reduce — Andrey Grozin.

Scilab — Serge Steer and Claude Gomez.

Shell — Joris van der Hoeven.

XYpic — Nicolas Ratier.

Yacas — Ayal Pinkus.

B.3.3. Administration of TEXMACS and material support

- Rennes Métropôle and the C.N.R.S. for financially supporting the development of TEXMACS.
- Christoph Benzmueller and his team for financially supporting the development of TEXMACS.
- Springer-Verlag for their financial support for making a better Windows version.
- Jean-Claude Fernandez, Fabien Salvi and the other persons from the CRI host and administrate the TEXMACS website.
- Álvaro Tejero Cantero maintains up the TEXMACS Wiki.
- Loic Dachary made TEXMACS accessible on Savannah.

B.3.4. Porting TEXMACS to other platforms

- Dan Martens is working on a the experimental Windows port.

- Marciano Siniscalchi ported TeXmacs to Cygwin. His work was further perfected by Loïc Pottier. Andreas Seidl made a the standard Cygwin package.
- Martin Costabel ported TeXmacs to MacOSX.
- Ralf Treinen and others has been ensuring the portability of TeXmacs to all architectures supported by Debian Gnu/Linux.
- Bruno Haible and Gregory Wright helped with porting TeXmacs to the SUN system and maintaining it.
- Philipp Tomsich and Chuck Sites for their help with the IRIX port.

B.3.5. Contributors to TeXmacs packages

- Ralf Treinen maintains the Debian package for TeXmacs.
- Christophe Merlet and Bo Forslund helped with making a portable RPM package.
- Lenny Cartier maintains the TeXmacs RPM for Mandrake Cooker.
- Jean Pierre Demailly and Yves Potin made TeXmacs part of the CNDP project to support free software.

B.3.6. Internationalization of TeXmacs

Czech. David Rezac.

Danish. Magnus Marius Rohde.

Dutch. Joris van der Hoeven.

Finnish. Teemu Ikonen.

French. Michèle Garoche, Joris van der Hoeven.

German. Dietmar Jung, Hans Dembinski, Jan Ulrich Hasecke, Christoph Strobel, Joris van der Hoeven, Thomas Langen, Ralf Treinen.

Hungarian. András Kadinger.

Italian. Andrea Centomo, Lucia Gecchelin, Xav and Daniele Pighin, Gian Luigi Gragnani.

Polish. Robert Janusz, Emil Nowak, Jan Alboszta.

Portuguese. Ramiro Brito Willmersdorf, Márcio Laurini, Alexandre Taschetto de Castro.

Romanian. Dan Ignat.

Russian. Andrey Grozin.

Slovene. Ziga Kranjec.

Spanish. Álvaro Cantero Tejero, Pablo Ruiz Múzquiz, David Moriano Garcia, Offray Vladimir Luna Cárdenas.

Swedish. Harald Ellmann.

Ukrainian. Volodymyr Lisivka.

B.3.7. Other contributors

Final thanks go to all others who have contributed to $\mathrm{T_EX_{MACS}}$, for instance by sending bug reports or by giving suggestions for future releases: Alexandre Abbes, Alessio Abogani, Aaron Acton, Till Adam, Murali Agastya, Guillaume Allègre, Larry D'Anna, Eizo Akiyama, Javed Alam, Doublet Alban, Michele Alessandrin, Andreas Almroth, Tom Alsberg, James Amundson, Piero D'Ancona, Daniel Andor, Ayal Anis, Javier Arantegui Jimenez, André Arnold, Uwe Assmann, Philippe Audebaud, Daniel Augot, Olaf Bachmann, Franky Backeljauw, Nick Bailey, Adrian Soto Banuelos, Pierre Barbier de Reuille, Marc Barisch, Giovanni Maniscalco Basile, Claude Baudouin, Marten Bauer, Luc Béhar, Roman Belenov, Odile Bénassy, Paul Benham, Roy C. Bentley, Attila Bergou, Christophe Bernard, Konrad Bernloehr, Karl Berry, Matthias Berth, Matteo Bertini, Cédric Bertolini, Matthew Bettencourt, Raktim Bhattacharya, Anne-Laure Biolley, Benedikt Birkenbach, Giovanni Biczó, Jim Blandy, Sören Blom, Christof Boeckler, François Bochatay, Christof Boeckler, Anton Bolfing, Robert Borys, Didier Le Botlan, Mohsen Bouaissa, Thierry Bouche, Adrien Bourdet, Michel Brabants, Didier Bretin, Jean-Yves Briend, Henrik Brink, Simon Britnell, Alexander M. Budge, Daniel Bump, Yoel Callev, José Cano, Charles James Leonardo Quarra Cappiello, Patrick Cardona, Niclas Carlsson, Dominique Caron, António Carvalho, Michel Castagner, Topher Cawlfield, Carlo Cecati, Beni Cherniavsky, Kuo-Ping Chiao, Teddy Fen-Chong, Henri Cohen, Johann Cohen-Tanugi, Vincenzo Colosimo, Dominique Colnet, Claire M. Connelly, Christoph Conrad, Riccardo Corradini, Paulo Correia, Olivier Cortes, Robert J. Cristel, Maxime Curioni, Allan Curtis, Jason Dagit, Stefano Dal Pra, François Dausseur, Thierry Dalon, Jon Davidson, Mike Davidson, Thomas Delzant, Jean-Pierre Demailly, Peter Denisevich, Alessio Dessi, Benno Dielmann, Lucas Dixon, Mikael Djurfeldt, Gabriel Dos Reis, Alban Doublet, Steingrim Dovland, Michael John Downes, Benjamin Drieu, Jose Duato, Amit Dubey, Daniel Duparc, Guillaume Duval, Tim Ebringer, Dirk Eddelbuettel, Magnus Ekdahl, Ulf Ekström, Sreedhar Ellisetty, Luis A. Escobar, Thomas Esser, Stephan Fabel, Robin Fairbairns, Tony Falcone, Vladimir Fedonov, Hilaire Fernandes, Ken Feyl, Juan Flynn, Jens Finke, Thomas Fischbacher, Cedric Foellmi, Enrico Forestieri, Ted Forringer, Christian Forster, Charlie Fortner, Stefan Freinatis, Michael P Friedlander, Nils Frohberg, Rudi Gaelzer, Maciej Gajewski, Lionel Garnier, Philippe Gogol, Björn Gohla, Patrick Gonzalez, Nirmal Govind, Albert Graef, Michael Graffam, Klaus Graichen, Ian Grant, Frédéric Grasset, Guido Grazioli, Wilco Greven, Cyril Grunspan, Laurent Guillon, Yves Guillou, Tae-Won Ha, Harri Haataja, Sébastien Hache, Irwan Hadi, James W. Haefner, Sam Halliday, Ola Hamfors, Aaron Hammack, Guillaume Hanrot, Alexander K. Hansen, Peter I. Hansen, Zaid Harchaoui, Jesper Harder, Philipp Hartmann, P. L. Hayes, Karl M. Hegbloom, Jochen Heinloth, Gunnar Hellmund, Ralf Hemmecke, Roy Henk, John Hernlund, Alain Herreman, Alexander Heuer, Johannes Hirn, Santiago Hirschfeld, Andreas Horn, Peter Horn, Chu-Ching Huang, Sylvain Huet, Ed Hurst, Karl Jarrod Hyder, Richard Ibbotson, Benjamin T. Ingram, Alexander Isacson, Michael Ivanov, Vladimir G. Ivanovic, Maik Jablonski, Frederic de Jaeger, Pierre Jarillon, Neil Jerram, Paul E. Johnson, Pierre-Henri Jondot, Peter Jung, Antoun Kanawati, Tim Kaulmann, Mukund S. Kalisi, Antoun Kanawati, Yarden Katz, Bernhard Keil, Samuel Kemp, Jeremy Kephart, Michael Kettner, Salman Khilji, Iwao

Kimura, Simon Kirkby, Ronny Klein, Peter Koepke, Matthias Koeppe, John Kollar, Denis Kovacs, Jeff Kowalczyk, Dmitri Kozionov, Ralph Krause, Neel Krishnaswami, Anthony Lander, Friedrich Laher, Winter Laite, Anthony Lander, Russell Lang, David Latreyte, Christopher Lee, Milan Lehocky, Torsten Leidig, Patrick Lenz, Kalle Lertola, Tristan Ley, Joerg Lippmann, Marc Longo, Pierre Lorenzon, Ralph Lõvi, V. S. Lugovsky, Gregory Lussiana, Bud Maddock, Duraid Madina, Camm Maguire, Yael Maguire, Paul Magwene, Jeremiah Mahler, Vincent Maillot, Giacomo Mallucci, Lionel Elie Mamane, Sourav K. Mandal, Andy P. Manners, Yun Mao, Chris Marcellin, Sylvain Marchand, Bernd Markgraf, Eric Marsden, Chris Marston, Evan Martin, Carlos Dehesa Martínez, Paulo Jorge de Oliveira Cantante de Matos, Tom McArdell, Bob McElrath, Alisdair McDiarmid, Robert Medeiros, Phil Mendelsohn, Sébastien de Menten, Jean-Michel Mermet, Jon Merriman, Herve le Meur, Ingolf Meyer, Amir Michail, Franck Michel, Jan David Mol, Klaus-Dieter Möller, Juan Fresneda Montano, André Moreau, Vijayendra Munikoti, Arkadiusz Miskiewicz, Sasha Mitelman, Dirk Moebius, Jack Moffitt, Harvey Monder, Guillaume Morin, Julian Morrison, Bernard Mourrain, Stephan Mucha, Toby Muhlhofer, Nathan Myers, Norbert Nemec, Thomas Neumann, Thien-Thi Nguyen, Han-Wen Nienhuys, Nix N. Nix, Eduardo Nogueira, Immanuel Normann, Jean-Baptiste Note, Ralf Nuetzel, Kostas Oikonomou, Ondrej Pacovsky, Bill Page, Santtu Pajukanta, Pierre Pansu, Ilya Papiashvili, Bernard Parisse, Frédéric Parrenin, André Pascual, Fernández Pascual, Frédéric Parrenin, Yannick Patois, Alen L. Peacock, François Pellegrini, Antonio Costa Pereira, Enrique Perez-Terron, Jacob Perkins, Bernard Perrot, Jan Peters, Jean Peyratout, Jacques Peyriere, Valery Pipin, Dimitri Pissarenko, Yves Pocchiola, Martin Pollet, Benjamin Poussin, Benjamin Podszun, Isaías V. Prestes, Rui Prior, Julien Puydt, Nguyen-Dai Quy, Manoj Rajagopalan, Ramakrishnan, Adrien Ramparison, Nicolas Ratier, Olivier Ravard, Leo Razoumov, Kenneth Reinhardt, Cesar A. Rendon, Diego Restrepo, Christian Requena, Chris Retford, Robert Ribnitz, Thomas CLive Richards, Staffan Ringbom, Eric Ringeisen, Christian Ritter, William G. Ritter, Will Robinson, Pascal Romon, Juan Pablo Romero, Juergen Rose, Mike Rosellini, Mike Rosing, Bernard Rousseau, Eyal Rozenberg, Olivier Ruatta, Filippo Rusconi, Gaetan Ryckeboer, Philippe Sam-Long, John Sandeman, Duncan Sands, Breton Saunders, Claire Sausset, David Sauzin, Gilles Schaeffer, Guido Schimmels, Rainer Schöpf, David Schweikert, Stefan Schwertheim, Rui Miguel Seabra, Chung-Tsun Shieh, Sami Sieranoja, Vasco Alexandre da Silva Costa, Marciano Siniscalchi, Daniel Skarda, Murray Smigel, Vaclav Smilauer, Dale P. Smith, Luke Snow, René Snyders, Pekka Sorjonen, Kasper Souren, Rodney Sparapani, Bas Spitters, Bas Spitters, Ivan Stanisavljevic, Starseeker, Harvey J. Stein, Peter Sties, Bernard Stloup, Peter Stoehr, Thierry Stoehr, James Su, Przemyslaw Sulek, Ben Sussman, Roman Svetlov, Milan Svoboda, Dan Synek, Pan Tadeusz, Luca Tagliacozzo, Sam Tannous, John Tapsell, Dung TaQuang, Gerald Teschl, Laurent Thery, Eric Thiébaut, Nicolas Thiery, Helfer Thomas, Reuben Thomas, Dylan Thurston, Kurt Ting, Janus N. Tøndering, Philippe Trébuchet, Marco Trevisani, Boris Tschirschwitz, Elias Tsigaridas, Michael M. Tung, Andreas Umbach, Miguel A. Valle, Rémi Vanicat, Harro Verkouter, Jacques Vernin, Sawan Vithlani, Philip A. Viton, Marius Vollmer, Guy Wallet, Adam Warner, Thomas Wawrzinek, Maarten Wegewijs, Duke Whang, Lars Willert, Grayson Williams, Barton Willis, Claus-Peter Wirth, Ben Wise, Wiebe van der Worp, Pengcheng Wu, Damien Wyart, Wang Yin, Lukas Zapletal, Volker Zell, Oleg Zhirov, Vadim V. Zhytnikov, Richard Zidlicky, Sascha Ziemann, Reinhard Zierke, Paul Zimmermann.

B.3.8. Contacting us

You can either contact us by email at

contact@texmacs.org

or by regular mail at

Joris van der Hoeven
Dépt. de Mathématiques (Bât. 425)
Université Paris-Sud
91405 Orsay Cedex
France

There are also several TeXmacs mailing lists:

texmacs-users@texmacs.org
texmacs-info@texmacs.org
texmacs-dev@gnu.org

B.4. Important changes in TeXmacs

Below, we briefly describe the most important changes which have occurred in TeXmacs since version 0.3.3.15. We also maintain a more detailed change log.

In general, when upgrading to a new version, we recommend you to make backups of your old TeXmacs files before opening them with the newer version of TeXmacs. In the unlikely case when your old file does not open in the correct way, please send a bug report to

bugs@texmacs.org

and send your old document as an attached file. Do not forget to mention your version of TeXmacs and the system you are using.

B.4.1. Improved titles (1.0.4.1)

From now on, titles of documents are more structured. This makes it easier to render the same title information in the appropriate ways for different styles. Old-style titles are automatically upgraded, but the result is only expected to be correct for documents with a single author. For documents with multiple authors, you may have to re-enter the title using our new interface.

B.4.2. Improved style sheets and source editing mode (1.0.3.5)

We are making it easier for users to edit style sheets. This improvement made it necessary to simplify many of the standard TeXmacs styles and packages, so that it will be easier to customize them. However, if you already designed some style files, then this may break some of their features. We mainly redesigned the list environments, the section environments and automatic numbering. Please report any problems to us.

B.4.3. Renaming of tags and environment variables (1.0.2.7 – 1.0.2.8)

Most environment variables and some tags have been renamed, so that these names no longer contain whitespace and only dashes (and no underscores) as separators.

B.4.4. Macro expansion (1.0.2.3 – 1.0.2.7)

An important internal change concerning the data format has been made: macro expansions and function applications like

```
(expand tag arg-1 ... arg-n)
```

```
(apply tag arg-1 ... arg-n)
```

are now replaced by hard-coded tags

```
(tag arg-1 ... arg-n)
```

Moreover, functions have systematically been replaced by macros. The few built-in functions which may take an arbitrary number of arguments have been rewritten using the new `xmacro` construct. If you ever wrote such a function yourself, then you will need to rewrite it too.

The new approach favorites a uniform treatment of macros and functions and makes the internal representation match with the corresponding SCHEME representation. More and more information about tags will gradually be stored in the D.R.D. (Data Relation Definition). This information is mostly determined automatically using heuristics.

Notice that some perverse errors might arise because of the above changes. Please keep copies of your old files and report any suspicious behaviour to us.

B.4.5. Formatting tags (1.0.2 – 1.0.2.1)

All formattings constructs without arguments (like line breaks, indentation directives, etc.) have been replaced by tags of arity zero. This makes most new documents badly unreadable for older versions of TEXMACS and subtle errors might occasionnaly occur when saving or loading, or during other editing operations.

B.4.6. Keyboard (1.0.0.11 – 1.0.1)

The TEXMACS keybindings have been rationalized. Here follows a list of the major changes:

- The `E-` prefix has been renamed to `M-`.
- `escape` is equivalent to `M-` and `escape`-`escape` to `A-`.
- Mode dependent commands are now prefixed by `A-`. In particular, accents are typed using `A-` instead of `E-`.
- Variants are now obtained using `tab` instead of `*` and you can circle back using `shift-tab`.
- Greek characters are now typed using `A-C-`, `F5`, or the hyper modifier, which can be configured in Edit → Preferences. You may also obtain Greek characters as variants of Latin characters. For instance, `p tab` yields π.
- The signification of the cursor keys in combination with control, alt and meta has changed.

You may choose between several "look and feels" for the keyboard behaviour in Edit → Preferences → Look and feel. The default is Emacs, but you may choose Old style if you want to keep the behaviour to which you may be used now.

B.4.7. Menus (1.0.0.7 – 1.0.1)

Several changes have been made in the menus. Here follows a list of the major changes:

- Buffer has been renamed as Go.
- Several items from File have been moved to View.
- The Edit → Import and Edit → Export items have been moved to Tools → Selections.
- The Insert menu has been split up into the menus Insert, Text and Mathematics.
- The Text and Paragraph menus have been merged together in one Format menu.
- Options has been spread out across Document, View, Tools and Edit → Preferences.

B.4.8. Style files (1.0.0.4)

Many changes have been made in the organization of the TeXmacs style files. Personal style files which depend on intermediate TeXmacs packages may require some slight adaptations.

We are working towards a stabilization of the standard style files and packages. At the end of this process, it should be easy to adapt existing LaTeX style files for journals to TeXmacs by customizing these standard style files and packages. As soon as we have time, we plan to provide online documentation on how to do this at Help → Online documentation.

B.4.9. Tabular material (0.3.5)

The way tabular material is treated has completely changed. It has become much easier to edit tables, matrices, equation arrays, etc. Also, many new features have been implemented, such as background color, border, padding, hyphenation, subtables, etc. However, the upgrading of old tabular material might sometimes be erroneous, in which case we invite you to submit a bug report.

B.4.10. Document format (0.3.4)

The TeXmacs document format has profoundly changed in order to make TeXmacs compatible with XML in the future. Most importantly, the old style environments like

```
<assign|env|<environment|open|close>>,
```

which are applied via matching pairs <begin|env>text<end|env>, have been replaced by macros

```
<assign|env|<macro|body|open<body>close>>,
```

which are applied via single macro expansions <expand|env|text>. Similarly, matching pairs <set|var|val>text<reset|var> of environment variable changes are replaced by a <with|var|val|text> construct (close to XML attributes). From a technical point of view, these changes lead to several complications if the text body consists of several paragraphs. As a consequence, badly structured documents may sometimes display differently in the new version (although I only noticed one minor change in my own documents). Furthermore, in order to maintain the higher level of structure in the document, the behaviour of the editor in relation to multiparagraph environments has slightly changed.

Appendix C

Contributing to GNU TeXmacs

C.1. Use TeXmacs

One of the best ways to contribute to GNU TeXmacs is by using it a lot, talk about it to friends and collegues, and to report me about bugs or other unnatural behaviour. Please mention the fact that you wrote articles using TeXmacs when submitting them. You can do this by putting the made-by-TeXmacs tag somewhere inside your title using Text → Title → TeXmacs notice.

Besides these general (but very important) ways to contribute, your help on the more specific subjects below would be appreciated. Don't hesitate to contact us if you want to contribute to these or any other issues. In the Help menu you can find documentation about the source code of TeXmacs, its document format, how to write interfaces with other formats, and so on.

C.2. Making donations to the TeXmacs project

Making donations to TeXmacs through the SPI organization.

One very important way to support TeXmacs is by donating money to the project. TeXmacs is currently one of the projets of SPI (Software in the Public Interest; see `http://www.spi-inc.org`). You may make donations of money to TeXmacs via this organization, by noting on your check or e-mail for wire transfers that your money should go to the TeXmacs project. You may also make donations of equipment or services or donations through vendors. See the SPI website for more information. We will maintain a webpage with a list of donors soon (if you agree to be on the list).

Details on how to donate money.

To make a donation, write a check or money order to:

> *Software in the Public Interest, Inc.*

and mail it to the following address:

> Software in the Public Interest, Inc.
> P.O. Box 502761
> Indianapolis, IN 46250-7761
> United States

To make an electronic transfer (this will work for non-US too), you need to give your bank the routing number and account number as follows:

> The SPI bank account is at American Express Centurion Bank.
> `Routing Number: 124071889`
> `Account Number: 1296789`

Don't forget to note on your check or e-mail for wire transfers that the money should be spent on the TeXmacs projet. In addition you may specify a more specific purpose on which you would like us to spend the money. You may also contact us for a more detailed discussion on this issue.

Important notes.

Let the SPI Treasurer (`treasurer@spi-inc.org`) know if you have problems. When you have completed the electronic wire, please send a copy of the receipt to the above address so there is a copy of your donation. The copy you send to the treasurer is important. You may also want to contact the TeXmacs team in order to make sure that the money arrived on the TeXmacs account.

Note: *The SPI address and account numbers may change from time to time. Please do not copy the address and account numbers, but rather point to the page* `http://www.spi-inc.org/donations` *to ensure that donors will always see the most current information.*

Donations in Europe can be done through our partner in Germany, ffis e.V. If you are interested in using their bank account (to save international money transfer costs), please check the instructions on `http://www.ffis.de/Verein/spi-en.html`.

C.3. CONTRIBUTE TO THE GNU $\mathrm{T_EX_{MACS}}$ DOCUMENTATION

There is a high need for good documentation on $\mathrm{T_EX_{MACS}}$ as well as people who are willing to translate the existing documentation into other languages. The aim of this site is to provide high quality documentation. Therefore, you should carefully read the guide-lines on how to write such documentation.

C.3.1. Introduction on how to contribute

High quality documentation is both a matter of content and structure. The content itself has to be as pedagogic as possible for the targeted group of readers. In order to achieve this, you should not hesitate to provide enough examples and illustrative screen shots whenever adequate. Although the documentation is not necessarily meant to be complete, we do aim at providing relatively stable documentation. In particular, you should have checked your text against spelling errors. The more experimental documentation should be put in the `incoming` directory or on the $\mathrm{T_EX_{MACS}}$ Wiki.

It is also important that you give your documentation as much structure as possible, using special markup from the `tmdoc` style file. This structure can be used in order to automatically compile printable books from your documentation, to make it suitable for different ways of viewing, or to make it possible to efficiently search a certain type of information in the documentation. In particular, you should always provide copyright and license information, as well as indications on how to traverse your documentation, if it contains many files.

Warning C.1. Don't forget to select Document → Language → Your language for each translated file. This will cause some content to be translated automatically, like the menus or some names of keys. Also, we recommend to run the TeXmacs spell checker on each translated document; this also requires the prior selection of the right document language.

C.3.2. Using CVS

The present TeXmacs documentation is currently maintained on `texmacs.org` using CVS (Concurrent Version System). In order to contribute, you should first create an account as explained on

http://www.texmacs.org/tmweb/download/cvs.en.html

In fact, the CVS system is not ideal for our documentation purpose, because it is not very dynamic. In the future, we plan to create a dedicated publication website, which will allow you to save documents directly to the web. It should also allow the automatic conversion of the documentation to other formats, the compilation of books, etc.

C.3.3. Conventions for the names of files

Most documentation should be organized as a function of the topic in a directory tree. The subdirectories of the top directory are the following:

devel. Documentation for developers.

examples. Examples of TeXmacs documents.

incoming. Incoming documentation, which is still a bit experimental.

main. The main documentation.

meta. How to write documentation and the compilation of documentation.

Please try to keep the number of entries per directory reasonably small.

File names in the main directory should be of the form `type-name.language.tm`. In the other directories, they are of the form `name.language.tm`. Here `type` is a major indication for the type of documentation; it should be one of the following:

adv. Documentation for advanced users.

man. For inclusion in the TeXmacs manual.

tut. For inclusion in the TeXmacs tutorial.

You should try to keep the documentation on the same topic together, regardless of the type. Indeed, this allows you to find more easily all existing documentation on a particular topic. Also, it may happen that you want to include some documentation which was initially meant for the tutorial in the manual. The `language` in which is the documentation has been written should be a two letter code like `en`, `fr`, etc. The main `name` of your file should be the same for the translations in other languages. For instance, `man-keyboard.en.tm` should not be translated as `man-clavier.fr.tm`.

C.3.4. Copyright information & the Free Documentation License

All documentation on the `texmacs-doc` site falls under the GNU Free Documentation License. If you write documentation for TEXMACS on this site, then you have to agree that it will be distributed under this license too. The copyright notice

```
Permission is granted to copy, distribute and/or modify this
document under the terms of the GNU Free Documentation License,
Version 1.1 or any later version published by the Free Software
Foundation; with no Invariant Sections, with no Front-Cover
Texts, and with no Back-Cover Texts. A copy of the license
is included in the section entitled "GNU Free Documentation
License".
```

should be specified at the end of *each* file. This should be done inside the tmdoc-license macro, in a similar way as at the end of the present document. When automatically generating a printed book from several documentation files, this will enable us to include the license only once.

You keep (part of) the copyright of all documentation that you will write for TEXMACS on the official `texmacs-doc` site. When you or others make additions to (or modifications in, or translations of) the document, then you should add your own name (at an appropriate place, usually at the end) to the existing copyright information. The copyright notice should be specified using the tmdoc-copyright function just before the license information at the end of the document. The first argument of this function contains a year or a period. Each remaining argument indicates one of the copyright holders. When combining (pieces of) several documents into another one, you should merge the copyright holders. For cover information (on a printed book for instance), you are allowed to list only the principal authors, but a complete list should be given at a clearly indicated place.

C.3.5. Traversing the TEXMACS documentation

As a general rule, you should avoid the use of sectioning commands inside the TEXMACS documentation and try to write small help pages on well identified topics. At a second stage, you should write recursive "meta help files" which indicate how to traverse the documentation in an automatic way. This allows the reuse of a help page for different purposes (a printed manual, a web-oriented tutorial, etc.).

The `tmdoc` style provides three markup macros for indicating how to traverse documentation. The traverse macro is used to encapsulate regions with traversal information. The branch macro indicates a help page which should be considered as a subsection and the continue macro indicates a follow-up page. Both the branch and the continue macro take two arguments. The first argument describes the link and the second argument gives the physical relative address of the linked file.

Typically, at the end of a meta help file you will find several branch or continue macros, inside one traverse macro. At the top of the document, you should also specify a title for your document using the tmdoc-title macro. When generating a printed manual from the documentation, a chapter-section-subsection structure will automatically be generated from this information and the document titles. Alternatively, one might automatically generate additional buttons for navigating inside the documentation using a browser.

C.3.6. Using the tmdoc style

Besides the copyright information macros and traversal macros, which have been documented before, the tmdoc style comes with a certain number of other macros and functions, which you should use whenever appropriate:

key. This macro is used to indicate keyboard input like `C-x C-s`. The specialized macros kbd-gen, kbd-text, kbd-math, kbd-symb, kbd-big, kbd-large, kbd-ia, kbd-exec and kbd-table are used for keyboard input corresponding to a specific type of action or mode. For instance, kbd-math corresponds to keyboard shortcuts for mathematical operations, such as `A-f`, which starts a fraction.

menu. This function with an arbitrary number of arguments indicates a menu like File or Document → Language. Menu entries are automatically translated by this function.

markup. This macro is used in order to indicate a macro or a function like section.

tmstyle. This macro indicates the name of a $\mathrm{T_EX_{MACS}}$ style file or package like `article`.

tmpackage. This macro indicates the name of a $\mathrm{T_EX_{MACS}}$ package like `std-markup`.

tmdtd. This macro indicates the name of a $\mathrm{T_EX_{MACS}}$ d.t.d. like `number-env`.

Notice that the contents of none of the above tags should be translated into foreign languages. Indeed, for menu tags, the translations are done automatically, so as to keep the translations synchronized with the translations of the actual $\mathrm{T_EX_{MACS}}$ menus. In the cases of markup, styles, packages and d.t.d.s, it is important to keep the original name, because it often corresponds to a file name.

The following macros and functions are used for linking and indexing purposes, although they should be improved in the future:

simple-link. This macro takes an URL x as argument and is a hyperlink with name and destination x.

hyper-link. This macro is a usual hyperlink.

concept-link. This macro takes a concept as argument. Later on an appropriate hyperlink might be created automatically from this and the other documentation.

only-index. Index a simple string.

def-index. Definition of a new concept; the text is printed in italic and indexed.

re-index. Reappearance of an already defined concept; the text is printed in roman and put in the index.

The following tags are also frequently used:

icon. Link to an icon in a central directory like `$TEXMACS_PATH/doc/images/pixmaps`.

screenshot. Link to a screenshot. The actual screenshots are stored in a central directory like `$TEXMACS_PATH/doc/images/screenshots`.

scheme. The SCHEME language.

cpp. The C++ language.

framed-fragment. For displaying a piece of code in a nice frame.

scheme-fragment. For multi-paragraph SCHEME code.

cpp-fragment. For multi-paragraph C++ code.

tm-fragment. For a piece of TEXMACS markup code in SCHEME format.

scheme-code. For a short piece of SCHEME code.

cpp-code. For a short piece of C++ code.

descriptive-table. For descriptive tables; such tables can be used to document lists of keyboard shortcuts, different types of markup, etc.

The `tmdoc` style inherits from the `generic` style and you should use macros like em, verbatim, itemize, etc. from this style whenever appropriate.

C.4. INTERNATIONALIZATION

The support of a maximal number of foreign languages is another major challenge in which your help would be appreciated. Making the translations to support a new language usually requires several days of work. We therefore recommend you to find some friends or collegues who are willing to help you.

The procedure for adding a new language is as follows

- You copy the file `english-new.scm` to `english-`*`yourlanguage`*`.dic` in `langs/natural/dic` and fill out the corresponding translations. You may want to use Andrey Grozin's dictionary tool at

  `http://www.texmacs.org/Data/dictool.py.gz`

  In order to use it, may sure that Python is installed on your system, download the file, gunzip it, make it executable and run it.

- You tell me about any special typographical rules in your language and handy keystrokes for producing special characters.

- I take care of the hyphenation and typographical issues, but you test them.

- If you have enough time, you may also consider the translation of (part of) the existing documentation.

Of course, the support for languages get out of date each time that new features are added to TEXMACS. For this reason, we also maintain a file `miss-english-`*`yourlanguage`*`.dic` with all missing translation for your language, once that it has been added. Please do not hesitate to send inclomplete versions of `english-`*`yourlanguage`*`.dic` or `miss-english-`*`yourlanguage`*`.dic`; someone else may be willing to complete them.

C.5. Writing data converters

If you are familiar with TeX, LaTeX, Html, Xml, Sgml, Mathml, Pdf, Rtf, or any other frequently used data format, please consider contributing to writing good converters for one or more of these formats. In Help → Source code → Data format you will find details about the TeXmacs data format and in Help → Source code → Data conversion we give some suggestions which might be helpful for these projects.

C.6. Porting TeXmacs to other platforms

Currently, TeXmacs is supported on most major Unix/X-Window platforms and a Windows port should be ready soon. Nevertheless, your help is appreciated in order to keep the existing ports working. Some remaining challenges for porting TeXmacs are:

- A native port for MacOS-X.
- Ports to PDAs, first of all those which run Linux. It should be noticed that, with the current support for Freetype, TeXmacs no longer depends on TeX/LaTeX for its fonts. We expect it to be possible to obtain a reasonable ports for TeXmacs on PDAs with 32Mb and at least 100MHz clock-speed. Of course, one also needs to customize the menus and/or icon bars, but this should not be hard.

  TeXmacs ports to PDAs would be particularly interesting in combination with the available plug-ins for doing scientific computations.

C.7. Interfacing TeXmacs with other systems

It is quite easy to write interfaces between TeXmacs and computer algebra systems or other scientific programs with structured output. Please consider writing interfaces between TeXmacs and your favorite system(s). TeXmacs has already been interfaced with several other free systems, like Giac, Macaulay 2, Maxima, GNU Octave, Pari, Qcl, gTybalt, Yacas. Detailed documentation on how to add new interfaces is available in the Help → Interfacing menu.

C.8. TeXmacs over the network and over the web

With the current technology of mutator tags, it should be quite easy to write a plug-in for TeXmacs for doing instant messenging or live-conferencing. We are very interested in people who would like to help with this. The same techniques might be used for collaborative authoring and educational purposes.

Besides live conferencing, we are also interested by people who are willing to program better integration of TeXmacs with the web. As a first step, this would require an internal C++ plug-in based on Wget or Curl for accessing web-pages, which supports cookies, security, etc. At a second stage, these features should be exploited by the Html converters. At the last stage, one might develop more general web-based services.

C.9. Become a TeXmacs developer

Apart from the kind of contributions which have been described in more detail above, there are many more issues where your help would be appreciated. Please take a look at our plans for the future for more details. Of course, you should feel free to come up with your own ideas and share them with us on the `texmacs-dev@gnu.org` mailing list!

Appendix D
Interfacing TeXmacs with other programs

D.1. Introduction

In this chapter we describe how to interface TeXmacs with an extern application. Such interfaces should be distributed in the form of plugins. The plug-in may either contain the extern application, or provide the "glue" between TeXmacs and the application. Usually, interfaces are used interactively in shell sessions (see Text → Session). But they may also be designed for background tasks, such as spell checking or typesetting.

The communication between TeXmacs and the application takes place using a customizable input format and the special *TeXmacs meta-format* for output from the plug-in. The meta-format enables you to send structured output to TeXmacs, using any common format like `verbatim`, LaTeX, PostScript, HTML, or TeXmacs itself. This is useful when adding a TeXmacs interface to an existing system, since LaTeX or PostScript output routines are often already implemented. It will then suffice to put the appropriate markers in order to make a first interface with TeXmacs.

As soon as basic communication between your application and TeXmacs is working, you may improve the interface in many ways. Inside shell sessions, there is support for prompts, default inputs, tab-completion, mathematical and multi-line input, etc. In general, your application may take control of TeXmacs and modify the user interface (menus, keyboard, etc.) or add new Scheme routines to TeXmacs. Your application may even extend the typesetter.

In the directory `$TEXMACS_PATH/examples/plugins`, you can find many examples of simple plug-ins. In the next sections, we will give a more detailed explanation of the interfacing features of TeXmacs on the hand of these examples. In order to try one of these examples, we recall that you just have to copy it to either one of the directories

`$TEXMACS_PATH/plugins`

`$TEXMACS_HOME_PATH/plugins`

and run the `Makefile` (if there is one).

D.2. Basic input/output using pipes

The configuration and the compilation of the `minimal` plug-in is described in the chapter about plug-ins. We will now study the source file `minimal/src/minimal.cpp`. Essentially, the `main` routine is given by

```
int
main () {
  display-startup-banner
  while (true) {
    read-input
    display-output
  }
  return 0;
}
```

By default, TeXmacs just send a `'\n'`-terminated string to the application as the input. Consequently, the code for ***read-input*** is given by

```
char buffer[100];
cin.getline (buffer, 100, '\n');
```

The output part is more complicated, since TeXmacs needs to have a secure way for knowing whether the output has finished. This is accomplished by encapsulating each piece of output (in our case both the display banner and the interactive output) inside a block of the form

```
DATA_BEGIN format:message DATA_END
```

Here `DATA_BEGIN` and `DATA_END` stand for special control characters:

```
#define DATA_BEGIN   ((char) 2)
#define DATA_END     ((char) 5)
#define DATA_ESCAPE  ((char) 27)
```

The `DATA_ESCAPE` is used for producing the `DATA_BEGIN` and `DATA_END` characters in the ***message*** using the rewriting rules

DATA_ESCAPE	DATA_BEGIN	⟶	DATA_BEGIN
DATA_ESCAPE	DATA_END	⟶	DATA_END
DATA_ESCAPE	DATA_ESCAPE	⟶	DATA_ESCAPE

The ***format*** specifies the format of the ***message***. For instance, in our example, the code of ***display-startup-banner*** is given by

```
cout << DATA_BEGIN << "verbatim:";
cout << "Hi there!";
cout << DATA_END;
fflush (stdout);
```

Similarly, the code of ***display-output*** is given by

```
cout << DATA_BEGIN << "verbatim:";
cout << "You typed " << buffer;
cout << DATA_END;
fflush (stdout);
```

REMARK D.1. For synchronization purposes, TeXmacs will assume that the output is finished as soon as it encounters the `DATA_END` which closes the initial `DATA_BEGIN`. So all output has to be inside one *single* outer `DATA_BEGIN`-`DATA_END` block: if you send more blocks, then TeXmacs will retake control before reading all your output. It *is* possible to nest `DATA_BEGIN`-`DATA_END` blocks though, as we will see below.

REMARK D.2. In our example, the C++ code for the application is included in the plug-in. In the case when you are writing a TeXmacs interface for an existing application ***myapp***, the convention is to create a `--texmacs` option for this program. Then it is no longer necessary to have ***myapp***`/src` and ***myapp***`/bin` directories for your plug-in and it suffices to configure the plug-in by putting something like the following in ***myapp***`/progs/init-`***myapp***`.scm`:

```
(plugin-configure myapp
  (:require (url-exists-in-path? "myapp"))
  (:launch "myapp --texmacs")
  (:session "Myapp"))
```

In the case when you do not have the possibility to modify the source code of ***myapp***, you typically have to write an input/output filter `tm_myapp` for performing the appropriate rewritings. By looking at the standard plug-ins distributed with TeXmacs in

`$TEXMACS_PATH/plugins`

you can find several examples of how this can be done.

D.3. FORMATTED AND STRUCTURED OUTPUT

In the previous section, we have seen that output from applications is encapsulated in blocks of the form

```
DATA_BEGIN format:message DATA_END
```

In fact, the ***message*** may recursively contain blocks of the same form. Currently implemented formats include `verbatim`, `latex`, `html`, `ps`, `scheme`. The `scheme` format is used for sending TeXmacs trees in the form of SCHEME expressions.

The `formula` plug-in.

The `formula` plug-in demonstrates the use of LaTeX as the output format. It consists of the files

`formula/Makefile`

`formula/progs/init-formula.scm`

`formula/src/formula.cpp`

The body of the main loop of `formula.cpp` is given by

```
int i, nr;
cin >> nr;
cout << DATA_BEGIN << "latex:";
cout << "$";
for (i=1; i<nr; i++)
  cout << "x_{" << i << "}+";
cout << "x_{" << i << "}$";
cout << DATA_END;
fflush (stdout);
```

Similarly, the use of nested output blocks is demonstrated by the `nested` plug-in; see in particular the source file `nested/src/nested.cpp`.

REMARK D.3. At the moment, we only implemented LaTeX as a standard transmission format for mathematical formulas, because this is the format which is most widely used. In the future, we intend to implement more semantically secure formats, and we recommend you to keep in mind the possibility of sending your output in tree format.

Nevertheless, we enriched standard LATEX with the * and \bignone commands for multiplication and closing big operators. This allows us to distinguish between

```
a \* (b + c)
```

(i.e. a multiplied by $b+c$) and

```
f(x + y)
```

(i.e. f applied to $x+y$). Similarly, in

```
\sum_{i=1}^m a_i \bignone + \sum_{j=1}^n b_j \bignone
```

the \bignone command is used in order to specify the scopes of the \sum operators.

It turns out that the systematic use of the * and \bignone commands, in combination with clean LATEX output for the remaining constructs, makes it *a priori* possible to associate an appropriate meaning to your output. In particular, this usually makes it possible to write additional routines for copying and pasting formulae between different systems.

The markup plug-in.

It is important to remind that structured output can be combined with the power of TEXMACS as a structured editor. For instance, the markup plug-in demonstrates the definition of an additional tag foo, which is used as an additional primitive in the output of the application. More precisely, the markup plug-in consists of the following files:

markup/Makefile

markup/packages/session/markup.ts

markup/progs/init-markup.scm

markup/src/markup.cpp

The style package markup.ts contains the following definition for foo:

```
⟨with|mode|math|⟨assign|foo|⟨macro|x|⟨frac|1|1+x⟩⟩⟩⟩
```

The foo tag is used in the following way in the body of the main loop of markup.cpp:

```
char buffer[100];
cin.getline (buffer, 100, '\n');
cout << DATA_BEGIN << "latex:";
cout << "$\\foo{" << buffer << "}$";
cout << DATA_END;
fflush (stdout);
```

Notice that the style package markup.ts also defines the markup-output environment:

```
⟨assign|markup-output|⟨macro|body|⟨generic-output|⟨with|par-mode|center|
body⟩⟩⟩⟩
```

This has the effect of centering the output in sessions started using Text → Session → Markup.

D.4. Output channels, prompts and default input

Besides blocks of the form

```
DATA_BEGIN format:message DATA_END
```

the TeXmacs meta-format also allows you to use blocks of the form

```
DATA_BEGIN channel#message DATA_END
```

Here ***channel*** specifies an "output channel" to which the body ***message*** has to be sent. The default output channel is `output`, but we also provide channels `prompt` and `input` for specifying the prompt and a default input for the next input in a session. Default inputs may be useful for instance be useful for demo modes of computer algebra systems. In the future, we also plan to support `error` and `status` channels.

The `prompt` plug-in.

The `prompt` plug-in shows how to use prompts. It consists of the files

`prompt/Makefile`

`prompt/progs/init-prompt.scm`

`prompt/src/prompt.cpp`

The routine for displaying the next prompt is given by

```
void
next_input () {
  counter++;
  cout << DATA_BEGIN << "prompt#";
  cout << "Input " << counter << "] ";
  cout << DATA_END;
}
```

This routine is both used for displaying the startup banner

```
cout << DATA_BEGIN << "verbatim:";
cout << "A LaTeX -> TeXmacs converter";
next_input ();
cout << DATA_END;
fflush (stdout);
```

and in the body of the main loop

```
char buffer[100];
cin.getline (buffer, 100, '\n');
cout << DATA_BEGIN << "verbatim:";
cout << DATA_BEGIN;
cout << "latex:$" << buffer << "$";
cout << DATA_END;
next_input ();
cout << DATA_END;
fflush (stdout);
```

D.5. SENDING COMMANDS TO TEXMACS

The application may use `command` as a very particular output format in order to send SCHEME commands to TEXMACS. In other words, the block

```
DATA_BEGIN command:cmd DATA_END
```

will send the command *cmd* to TEXMACS. Such commands are executed immediately after reception of `DATA_END`. We also recall that such command blocks may be incorporated recursively in larger `DATA_BEGIN`-`DATA_END` blocks.

The menus plug-in.

The `nested` plug-in shows how an application can modify the TEXMACS menus in an interactive way. The plug-in consists of the files

menus/Makefile

menus/progs/init-menus.scm

menus/src/menus.cpp

The body of the main loop of `menus.cpp` simply contains

```
char buffer[100];
cin.getline (buffer, 100, '\n');
cout << DATA_BEGIN << "verbatim:";
cout << DATA_BEGIN << "command:(menus-add \""
     << buffer << "\")" << DATA_END;
cout << "Added " << buffer << " to menu";
cout << DATA_END;
fflush (stdout);
```

The SCHEME macro `menus-add` is defined in `init-menus.scm`:

```
(menu-bind menus-menu
  ("Hi" (insert "Hello world")))

(menu-extend texmacs-extra-menu
  (if (equal? (get-env "prog language") "menus")
      (=> "Menus" (link menus-menu))))

(define-macro (menus-add s)
  `(menu-extend menus-menu
     (,s (insert ,s))))
```

The configuration of `menus` proceeds as usual:

```
(plugin-configure menus
  (:require (url-exists-in-path? "menus.bin"))
  (:launch "menus.bin")
  (:session "Menus"))
```

D.6. Background evaluations

Until now, we have always considered interfaces between TeXmacs and applications which are intended to be used interactively in shell sessions. But there also exists a Scheme command

```
(plugin-eval plugin session expression)
```

for evaluating an expression using the application. Here ***plugin*** is the name of the plug-in, ***session*** the name of the session and ***expression*** a Scheme expression which represents a TeXmacs tree.

The `substitute` plug-in.

Background evaluations may for instance be used in order to provide a feature which allows the user to select an expression and replace it by its evaluation. For instance, the `substitute` plug-in converts mathematical LaTeX expressions into TeXmacs, and it provides the `C-F12` keyboard shortcut for replacing a selected text by its conversion. The plug-in consists of the following files

`substitute/Makefile`

`substitute/progs/init-substitute.scm`

`substitute/src/substitute.cpp`

The main evaluation loop of `substitute.cpp` simply consists of

```
char buffer[100];
cin.getline (buffer, 100, '\n');
cout << DATA_BEGIN;
cout << "latex:$" << buffer << "$";
cout << DATA_END;
fflush (stdout);
```

Moreover, the configuration file `init-substitute.scm` contains the following code for replacing a selected region by its evaluation

```
(define (substitute-substitute)
  (import-from (texmacs plugin plugin-cmd))
  (if (selection-active-any?)
      (let* ((t (tree->stree (the-selection)))
             (u (plugin-eval "substitute" "default" t)))
        (clipboard-cut "primary")
        (insert (stree->tree u)))))
```

as well as the keyboard shortcut for `C-F12`:

```
(kbd-map
  ("C-F12" (substitute-substitute)))
```

Notice that these routines should really be defined in a separate module for larger plug-ins.

The `secure` plug-in.

Another example of using an interface in the background is the `secure` plug-in which consists of the files

`secure/Makefile`

`secure/packages/secure.ts`

`secure/progs/init-secure.scm`

`secure/progs/secure-secure.scm`

`secure/src/secure.cpp`

Just as `substitute.cpp` above, the main program `secure.cpp` just converts mathematical LATEX expressions to TEXMACS. The `secure-secure.scm` module contains the *secure* SCHEME routine `latexer`:

```
(tm-define (latexer s)
  (:type (tree -> object))
  (:synopsis "convert LaTeX string to TeXmacs tree using
plugin")
  (:secure #t)
  (plugin-eval "secure" "default" (tree->string s)))
```

It is important to define `latexer` as being secure, so that it can be used in order to define additional markup using the extern primitive. This is done in the style file `secure.ts`:

See a LaTeX math command as a TeXmacs expression via plug-in
⟨assign|*latexer*|⟨macro|*x*|⟨extern|latexer|*x*⟩⟩⟩

After compilation, installation, relaunching TEXMACS and selecting Document → Use package → secure, you will now be able to use latexer as a new primitive. The primitive takes a mathematical LATEX expression as its argument and displays its TEXMACS conversion.

D.7. MUTATOR TAGS

A mutator tag is of the form ⟨mutator|*body*|*cmd*⟩, where *body* is the visible body of the tag and *cmd* a secure SCHEME script which is called periodically and which is allowed to modify the *body* of the mutator (or even any other part of the document). During the execution of *cmd* the function `mutator-path` yields the position of the *body* in the document tree.

Mutator tags are a particularly interesting feature of TEXMACS for producing highly interactive documents. For instance, inside computer algebra sessions, the output is retrieved inside a mutator tag (which automatically removes itself when the output is complete). In the future, mutators might be used for live conferencing or in interfaces with proof systems.

REMARK D.4. Mutator tags only work properly when the explicitly occur in the document; they will not work if they merely occur in the body of a macro, even if the body of the mutator remains accessible.

Indeed, the current implementation of TEXMACS searches for mutator tags in all documents after a small period of inactivity. Documents which are known not to contain mutator tags are ignored during this research. Of course, this implementation is both quite efficient and incompatible with the macro system. So there is room for future improvements.

REMARK D.5. For efficiency reasons, it is recommended that mutator tags mainly modify there own bodies and not other parts of the document, except at really exceptional occasions.

The `mutator` plug-in.

A very simple example with two types of mutators is provided in the **`mutator`** plug-in. It provides the user with two keyboard shortcuts `C-F11` and `C-F12`, which respectively insert the current time and some blinking text. The plug-in consists of the single file

`mutator/progs/init-mutator.scm`

The `C-F11` key simply inserts ⟨mutator|text|(mutate-date)⟩ into the main text:

```
(kbd-map ("C-F11" (insert '(mutator "" "(mutate-date)"))))
```

The secure SCHEME routine **`mutate-date`** is defined as follows:

```
(tm-define (mutate-date)
  (:secure #t)
  (let* ((p (the-mutator-path))
         (date (var-eval-system "date +\"%H:%M:%S\"")))
    (tm-assign-diff p date)))
```

The **`tm-assign-diff`** command is convenient, because it only modifies the document if a real change occurred.

The insertion of blinking content is slightly more complex, since it also takes into account the current content of the mutator tag. The `C-F12` key inserts ⟨mutator|text|(mutate-blink)⟩ into the main text and puts the cursor after the text in the body of the mutator:

```
(kbd-map ("C-F12" (insert-go-to '(mutator "text" "(mutate-
blink)")
                                '(0 4))))
```

The secure SCHEME routine **`mutate-blink`** is defined as follows:

```
(tm-define (mutate-blink)
  (:secure #t)
  (let* ((mod (lambda (x y) (* y (- (/ x y) (floor (/ x
y))))))
         (p (the-mutator-path))
         (t (tm-subtree p))
         (s (string->number (var-eval-system "date +\"%S\"")))
         (e (mod s 4)))
    (if (and (<= e 1) (not (match? t '(strong :1))))
        (tm-ins-unary p 'strong))
    (if (and (>= e 2) (match? t '(strong :1)))
        (tm-rem-unary p))))
```

REMARK D.6. Notice that the above examples are only meant to illustrate the use of mutators. Ideally speaking, dates and blinking content should not make use of mutators, since mutators continuously *modify* the document (think about undoing changes, for instance). In the future, we plan to add primitives like animations and movies to $\text{T}_\text{E}\text{X}_{\text{MACS}}$, for which the content remains fixed, but whose presentation changes over time.

D.8. Mathematical and customized input

The TeXmacs meta-format allows application output to contain structured text like mathematical formulas. In a similar way, you may use general TeXmacs content as the input for your application. By default, only the text part of such content is kept and sent to the application as a string. Moreover, all characters in the range 0–31 are ignored, except for `'\t'` and `'\n'` which are transformed into spaces. There are two methods to customize the way input is sent to your application. First of all, the configuration option

```
(:serializer ,routine)
```

specifies a scheme function for converting TeXmacs trees to string input for your application, thereby overriding the default method. This method allows you for instance to treat multi-line input in a particular way or the perform transformations on the TeXmacs tree.

The `:serialize` option is a very powerful, but also a very abstract way to customize input: it forces you to write a complete input transformation function. In many circumstances, the user really wants to rewrite two dimensional mathematical input to a more standard form, like rewriting $\frac{a}{b}$ to `((a)/(b))`. Therefore, a second way for customizing the input is to use the command

```
(plugin-input-converters myplugin
  rules)
```

This command specifies input conversion rules for ***`myplugin`*** for "mathematical input" and reasonable defaults are provided by TeXmacs. Each rule is of one of the following two forms:

Leaf transformation rules.

Given two strings ***`symbol`*** and ***`conversion`***, the rule

```
(symbol conversion)
```

specifies that the TeXmacs symbol ***`symbol`*** should be converted to ***`conversion`***.

Tag transformation rules.

Given a symbol ***`tag`*** and a SCHEME function ***`routine`***, the rule

```
(tag routine)
```

specifies that ***`routine`*** will be used as the conversion routine for ***`tag`***. This routine should just write a string to the standard output. The SCHEME function `plugin-input` may be used for the recursive transformation of the arguments of the tag.

The `input` plug-in.

The `input` plug-in demonstrates the use of customized mathematical input. It consists of the files

`input/Makefile`

`input/packages/session/input.ts`

`input/progs/init-input.scm`

`input/progs/input-input.scm`

input/src/input.cpp

The Scheme configuration code in `init-input.scm` is given by

```
(plugin-configure input
  (:require (url-exists-in-path? "input.bin"))
  (:initialize (input-initialize))
  (:launch "input.bin")
  (:session "Input"))
```

Here `input-initialize` is an initialization routine which adds the new input conversion rules in a lazy way:

```
(define (input-initialize)
  (import-from (texmacs plugin plugin-convert))
  (lazy-input-converter (input-input) input))
```

In other words, the module `input-input.scm` will only be loaded when we explicitly request to make a conversion. The conversion rules in `input-input.scm` are given by

```
(plugin-input-converters input
  (frac input-input-frac)
  (special input-input-special)
  ("<vee>" "||")
  ("<wedge>" "&&"))
```

This will cause $\vee$ and $\wedge$ to be rewritten as `||` and `&&` respectively. Fractions $\frac{a}{b}$ are rewritten as `((a):(b))` using the routine

```
(define (input-input-frac t)
  (display "((")
  (plugin-input (car t))
  (display "):(")
  (plugin-input (cadr t))
  (display "))"))
```

In the additional style file `input.ts` we also defined some additional markup special:

```
⟨assign|special|
   ⟨macro|body|
      ⟨block|
         ⟨tformat|
            ⟨cwith|1|1|1|1|cell-background|pastel green⟩|
            ⟨table|
               ⟨row|⟨cell|body⟩⟩⟩⟩⟩⟩⟩
```

This tag is rewritten using the special conversion rule

```
(define (input-input-special t)
  (display "[[[SPECIAL:")
  (plugin-input (car t))
  (display "]]]"))
```

As to the C++ code in `input.cpp`, the startup banner automatically puts the shell session in mathematical input mode:

```
cout << DATA_BEGIN << "verbatim:";
cout << DATA_BEGIN << "command:(session-use-math-input #t)"
     << DATA_END;
cout << "Convert mathematical input into plain text";
cout << DATA_END;
fflush (stdout);
```

In the main loop, we content ourselves the reproduce the input as output:

```
char buffer[100];
cin.getline (buffer, 100, '\n');
cout << DATA_BEGIN << "verbatim:";
cout << buffer;
cout << DATA_END;
fflush (stdout);
```

D.9. TAB-COMPLETION

By default, TEXMACS looks into your document for possible tab-completions. Inside sessions for your application, you might wish to customize this behaviour, so as to complete built-in commands. In order to do this, you have to specify the configuration option

```
(:tab-completion #t)
```

in your `init-`*`myplugin`*`.scm` file, so that TEXMACS will send special tab-completion requests to your application whenever you press `tab` inside a session. These commands are of the form

```
DATA_COMMAND(complete input-string cursor-position)return
```

Here `DATA_COMMAND` stands for the special character `'\20'` (ASCII 16). The *`input-string`* is the complete string in which the `tab` occurred and the *`cursor-position`* is an integer which specifies the position of the cursor when you pressed `tab`. TEXMACS expects your application to return a tuple with all possible tab-completions of the form

```
DATA_BEGINscheme:(tuple root completion-1 ⋯ completion-
n)DATA_END
```

Here *`root`* corresponds to a substring before the cursor for which completions could be found. The strings *`completion-1`* until *`completion-n`* are the list of completions as they might be inserted at the current cursor position. If no completions could be found, then you may also return the empty string.

REMARK D.7. In principle, the tab-completion mechanism should still work in mathematical input mode. In that case, the *`input-string`* will correspond to the serialization of the TEXMACS input.

REMARK D.8. The way $\mathrm{T_EX_{MACS}}$ sends commands to your application can be customized in a similar way as for the input: we provide a `:commander` configuration option for this, which works in a similar way as the `:serializer` option.

The `complete` plug-in.

A very rudimentary example of how the tab-completion mechanism works is given by the `complete` plug-in, which consists of the following files:

complete/Makefile

complete/progs/init-complete.scm

complete/src/complete.cpp

The startup banner in `complete.cpp` takes care of part of the configuration:

```
cout << DATA_BEGIN << "verbatim:";
format_plugin ();
cout << "We know how to complete 'h'";
cout << DATA_END;
fflush (stdout);
```

Here `format_plugin` is given by

```
void
format_plugin () {
  // The configuration of a plugin can be completed at startup
time.
  // This may be interesting for adding tab-completion a
posteriori.
  cout << DATA_BEGIN << "command:";
  cout << "(plugin-configure complete (:tab-completion #t))";
  cout << DATA_END;
}
```

In the main loop, we first deal with regular input:

```
char buffer[100];
cin.getline (buffer, 100, '\n');
if (buffer[0] != DATA_COMMAND) {
  cout << DATA_BEGIN << "verbatim:";
  cout << "You typed " << buffer;
  cout << DATA_END;
}
```

We next treat the case when a tab-completion command is sent to the application:

```
else {
  cout << DATA_BEGIN << "scheme:";
  cout << "(tuple \"h\" \"ello\" \"i there\" \"ola\"
\"opsakee\")";
  cout << DATA_END;
}
fflush (stdout);
```

As you notice, the actual command is ignored, so our example is really very rudimentary.

D.10. Dynamic libraries

Instead of connecting your system to TeXmacs using a pipe, it is also possible to connect it as a dynamically linked library. Although communication through pipes is usually easier to implement, more robust and compatible with gradual output, the second option is faster.

In order to dynamically link your application to TeXmacs, you should follow the TeXmacs communication protocol, which is specified in the following header file:

`$TEXMACS_PATH/include/TeXmacs.h`

In this file it is specified that your application should export a data structure

```
typedef struct package_exports_1 {
  char* version_protocol; /* "TeXmacs communication protocol
1" */
  char* version_package;
  char* (*install) (TeXmacs_exports_1* TeXmacs,
                    char* options, char** errors);
  char* (*evaluate) (char* what, char* session, char**
errors);
} package_exports_1;
```

which contains an installation routine for your application, as well as an evaluation routine for further input (for more information, see the header file). TeXmacs will on its turn export a structure

```
typedef struct TeXmacs_exports_1 {
  char* version_protocol; /* "TeXmacs communication protocol
1" */
  char* version_TeXmacs;
} TeXmacs_exports_1;
```

It is assumed that each application takes care of its own memory management. Hence, strings created by TeXmacs will be destroyed by TeXmacs and strings created by the application need to be destroyed by the application.

The string `version_protocol` should contain `"TeXmacs communication protocol 1"` and the string `version_package` the version of your package. The routine `install` will be called once by TeXmacs in order to initialize your system with options `options`. It communicates the routines exported by TeXmacs to your system in the form of a pointer to a structure of type `TeXmacs_exports_1`. The routine should return a status message like

```
"yourcas-version successfully linked to TeXmacs"
```

If installation failed, then you should return `NULL` and `*errors` should contain an error message.

The routine `evaluate` is used to evaluate the expression `what` inside a TeXmacs-session with name `session`. It should return the evaluation of `what` or `NULL` if an error occurred. `*errors` either contains one or more warning messages or an error message, if the evaluation failed. The formats being used obey the same rules as in the case of communication by pipes.

Finally, the configuration file of your plug-in should contain something as follows:

```
(plugin-configure myplugin
  (:require (url-exists? (url "$LD_LIBRARY_PATH"
"libmyplugin.so")))
  (:link "libmyplugin.so" "myplugin_exports" "")
  further-configuration)
```

Here *myplugin*`_exports` is a pointer to a structure of the type `package_exports_1`.

Remark D.9. It is possible that the communication protocol changes in the future. In that case, the data structures `TeXmacs_exports_1` and `package_exports_1` will be replaced by data structures `TeXmacs_exports_n` and `package_exports_n`, where `n` is the version of the protocol. These structures will always have the abstract data structures `TeXmacs_exports` and `package_exports` in common, with information about the versions of the protocol, TeXmacs and your package.

The `dynlink` plug-in.

The `dynlink` plug-in gives an example of how to write dynamically linked libraries. It consists of the following files:

`dynlink/Makefile`

`dynlink/progs/init-dynlink.scm`

`dynlink/src/dynlink.cpp`

The `Makefile` contains

```
tmsrc = /home/vdhoeven/texmacs/src/TeXmacs
CXX = g++
LD  = g++

lib/libtmdynlink.so: src/dynlink.cpp
        $(CXX) -I$(tmsrc)/include -c src/dynlink.cpp -o
src/dynlink.o
        $(LD) -shared -o lib/libtmdynlink.so src/dynlink.o
```

so that running it will create a dynamic library `dynlink/lib/libdynlink.so` from `dynlink.cpp`. The `tmsrc` variable should contain `$TEXMACS_PATH`, so as to find the include file `TeXmacs.h`. The configuration file `init-dynlink.scm` simply contains

```
(plugin-configure dynlink
  (:require (url-exists? (url "$LD_LIBRARY_PATH"
                              "libtmdynlink.so")))
  (:link "libtmdynlink.so" "dynlink_exports" "")
  (:session "Dynlink"))
```

As to the C++ file `dynlink.cpp`, it contains a string

```
static char* output= NULL;
```

with the last output, the initialization routine

```
char*
dynlink_install (TeXmacs_exports_1* TM, char* opts, char**
errs) {
  output= (char*) malloc (50);
  strcpy (output, "\2verbatim:Started dynamic link\5");
  return output;
}
```

the evaluation routine

```
char*
dynlink_eval (char* what, char* session, char** errors) {
  free (output);
  output= (char*) malloc (50 + strlen (what));
  strcpy (output, "\2verbatim:You typed ");
  strcat (output, what);
  strcat (output, "\5");
  return output;
}
```

and the data structure with the public exports:

```
package_exports_1 dynlink_exports= {
  "TeXmacs communication protocol 1",
  "Dynlink 1",
  dynlink_install,
  dynlink_eval
};
```

Notice that the application takes care of the memory allocation and deallocation of `output`.

D.11. MISCELLANEOUS FEATURES

Several other features are supported in order to write interfaces between $\TeX_{\text{MACS}}$ and extern applications. Some of these are very hairy or quite specific. Let us briefly describe a few miscellaneous features:

Interrupts.

The "stop" icon can be used in order to interrupt the evaluation of some input. When pressing this button, $\TeX_{\text{MACS}}$ will just send a `SIGINT` signal to your application. It expects your application to finish the output as usual. In particular, you should close all open `DATA_BEGIN`-blocks.

Testing whether the input is complete.

Some systems start a multiline input mode as soon as you start to define a function or when you enter an opening bracket without a matching closing bracket. $\TeX_{\text{MACS}}$ allows your application to implement a special predicate for testing whether the input is complete. First of all, this requires you to specify the configuration option

```
(:test-input-done #t)
```

As soon as you will press `return` in your input, TeXmacs will then send the command

```
DATA_COMMAND (input-done? input-string) return
```

Your application should reply with a message of the form

```
DATA_BEGIN scheme:done DATA_END
```

where ***done*** is either `#t` or `#f`. The `multiline` plug-in provides an example of this mechanism (see in particular the file `multiline/src/multiline.cpp`).

D.12. Plans for the future

There are many improvements to be made in the TeXmacs interface to computer algebra systems. First of all, the computer algebra sessions have to be improved (better hyphenation, folding, more dynamic subexpressions, etc.).

As to interfaces with computer algebra systems, out main plans consist of providing tools for semantically safe communication between several system. This probably will be implemented in the form of a set of plug-ins which will provide conversion services.

Index