The next step is to make sure that TeXmacs has been installed on both computers. The remote TeXmacs installation will mainly be used in order to detect those plug-ins that can be used on the remote computer.

When everything has been set up in this way, select Insert ▸ Session ▸ Remote in order to open the remote plug-in selector. Add the name of the remote server by typing its name or IP address and clicking on Add. After a small pause, the remote server should appear in the list together with the remote plug-ins that are supported. You may now simply select the plug-in you want to use from the list. Notice that remote plug-ins may take a few seconds in order to boot.

Servers that have been added to the list of remote plug-in servers will be remembered the next time you start TeXmacs. You may use the buttons Remove and Update in order to remove a server from the list and to update the list of supported remote plug-ins.

Chapter 12
Get it your way

12.1 Creating macros

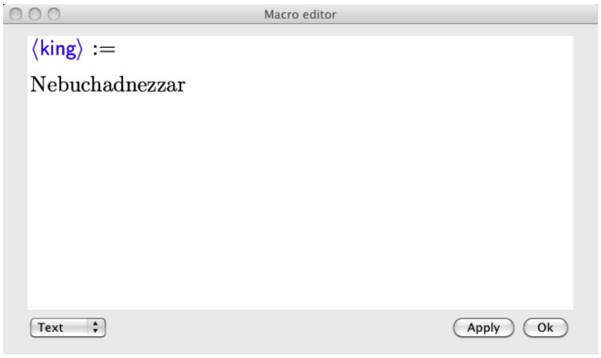
12.1.1 Hello Nebuchadnezzar

We have encountered an avalanche of markup elements throughout this book. Why add your own ones on top of the existing? One possible reason is that you may wish to save time by introducing abbreviations for lengthy names or notations. Let us see how to do this by defining your own *macros*.

Assume that we need the name of a king for our new novel and that Nebuchadnezzar seems to be the perfect candidate. Then it is natural to introduce a new macro king as an abbreviation for Nebuchadnezzar. The easiest way to do this is to open the "macro definition widget" using Tools ▸ Macros ▸ New macro. At the place of *enter-name*, you may enter the name of your macro: "king". The corresponding body "Nebuchadnezzar" can be typed below the := sign. When done (see Figure 12.1), simply click on Ok. Now that your macro has been defined, you can use it as many times as you wish by typing `\` `k` `i` `n` `g` `↩`:

> Hello Nebuchadnezzar!
> Nebuchadnezzar: my beloved leader.
> Bye, bye, Nebuchadnezzar.

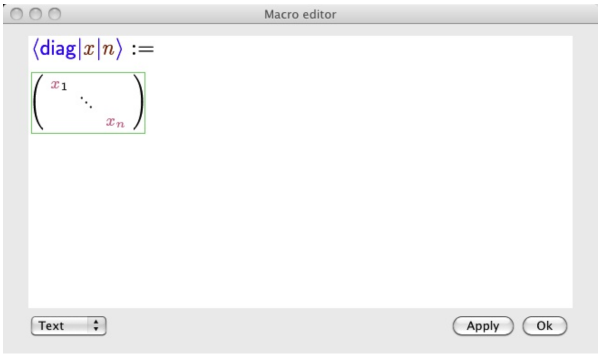
One big advantage of using macros for abbreviations and special notations is that it suffices to modify the macro definitions whenever you change your mind: by positioning your cursor right after one of the king tags, you can edit the corresponding macro using Focus ▸ Preferences ▸ Edit macro. Just change the body of the macro to "Arikesari Maravarman Nindraseer Nedumaaran", if this is your new hero, and all Nebuchadnezzars will mutate accordingly.


Macro editor

⟨king⟩ :=

Nebuchadnezzar

Figure 12.1. The widget for macro definitions.



$$\begin{pmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{pmatrix}$$

Figure 12.2. Example of a matrix definition with arguments.

12.1.2 Adding arguments

As our next example, assume that we wish to define a macro diag that produces a diagonal matrix

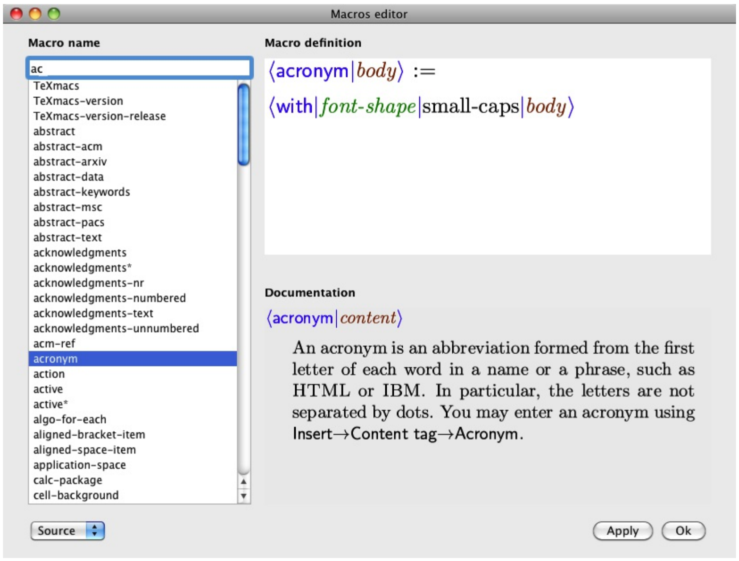
$$\begin{pmatrix} x_1 & & \\ & \ddots & \\ & & x_n \end{pmatrix}, \qquad (12.1)$$

with the additional constraint that the symbols x and n can be changed each time the macro is used.

After entering the name of the macro as in the previous example, you may use the ⌥→ shortcut in order to insert additional macro arguments, say x and n. When entering the body of the macro, these arguments can be used inside the body by typing \ x ↩ and \ n ↩. The resulting macro definition should look like Figure 12.2. When applying the macro using \ d i a g ↩, an empty and editable slot is provided for each argument. Notice that only the first top-left occurrence of the parameter x can be edited, although the second one at the bottom-right will be updated accordingly during modifications.

The widget for macro editing is mainly intended for quick definitions of simple macros. In particular, you may miss the menus and the toolbars of the usual editor. Nevertheless, most of the usual keyboard shortcuts do work, as well as the contextual menus that appear when pressing the right mouse button (if your mouse has a single button, then you should hold the control key ^ while pressing that button). Whenever some keyboard shortcuts are preempted by the operating system, we also recall from section 2.3.5 that you may emulate the modifier keys ^, ⌥, and ⌘ using ⎋. For instance, ⌥→ is equivalent to ⎋ ⎋ →.

You may have noticed that we used the \ key both for applying the diag macro and for inserting the macro arguments x and n. The behavior of the \ key is context-dependent indeed. Inside the definition of a macro with arguments x and n, typing \ x ↩ will lead to the insertion of the macro argument x. In a context where the diag macro is defined, TeXmacs will automatically apply this macro when typing \ d i a g ↩. Notice that macro arguments are indicated



ac

TeXmacs
TeXmacs-version
TeXmacs-version-release
abstract
abstract-acm
abstract-arxiv
abstract-data
abstract-keywords
abstract-msc
abstract-pacs
abstract-text
acknowledgments
acknowledgments*
acknowledgments-nr
acknowledgments-numbered
acknowledgments-text
acknowledgments-unnumbered
acm-ref
acronym
action
active
active*
algo-for-each
aligned-bracket-item
aligned-space-item
application-space
calc-package
cell-background

⟨acronym|*body*⟩ :=

⟨with|*font-shape*|small-caps|*body*⟩

⟨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 Insert→Content tag→Acronym.

Figure 12.3. Browsing the list of all macros using the macros editor.

using a brown color, whereas tags are displayed in blue.

Another technique that is useful for the definition of macros goes as follows. First design the prototype macro application in an ordinary TeXmacs window. That would simply be the diagonal matrix (12.1) for our example. Now copy this matrix and paste it inside the macro body when defining the diag macro. Finally replace the variables x and n by the macro arguments x and n that were entered using \ x ↩ and \ n ↩. This copy-and-paste technique will turn out to be even more powerful when editing style files, as described later in this chapter.

12.1.3 Customizing an existing macro

Whenever your cursor is inside or just after a markup element, you may customize the corresponding macro definition using Focus ▸ Preferences ▸ Edit macro or 🔧 ▸ Edit macro. For example, assume that we entered the acronym GNU or LASER (using \ a c r o n y m ↩ or Insert ▸ Content tag ▸ Acronym). If you wish to change the default rendering (a small capitals font), you can customize the corresponding acronym macro by positioning your cursor inside the text GNU or LASER and opening the macro editor with 🔧 ▸ Edit macro.

In certain cases, it may also be handy to browse the list of all existing macros that are active in your current style. This can be done by opening the macros editor via Tools ▸ Macros ▸ Edit macros: see Figure 12.3. You may find the acronym macro in the list at the left-hand side, possibly after typing the first few characters of the Macro name. A short description of the macro is displayed in the Documentation area, whenever available. The macro can be edited as usual in the Macro definition area.

In Figure 12.3, we selected "Source mode" for editing the body of the macro, as indicated by the pull-down menu at the bottom-left of the window. This mode is usually most convenient for editing complex macros, whereas the default text mode tends to be more suitable for simple, visually oriented macros. We also notice that the macro editor shows macro definitions in an abridged manner: inside style files and packages the definition from Figure 12.3 would read

```
⟨assign|acronym|⟨macro|body|⟨with|font-shape|small-caps|body⟩⟩⟩
```

Before we explain the presentation and syntax of source code in full detail, let us first analyze the customization of an existing macro on a more complex example.

12.1.4 Anatomy of a macro

The rendering of certain markup elements like acronym is quite straightforward. Many macros are far more complex and may recursively depend on other macros.

Assume for example that we wish to customize the proof environment using the macros editor and replace the "end-of-proof" symbol □ by ■. The internal structure of the proof is exposed by selecting Source instead of Text in the left bottom menu of the macros editor. For most styles, the source code is as follows:

```
⟨assign|proof|
    ⟨macro|body|
        ⟨render-proof|⟨proof-text⟩|body⟩⟩⟩
```

The prefix "render-" is used for submacros that specifically control the graphical layout. In this case, the arguments of render-proof are the (potentially translated) text "Proof" and the actual proof body. This additional level of indirection allows for the specification of alternative proof texts:

> **Continued proof of the master theorem.** We now conclude by applying the master lemma. □

When customizing a macro, it is important to first understand its structure and the intent of the other macros on which it relies. In many cases, you really want to customize one or more of these other macros. In our example, this leads us to examine the source code of the render-proof macro, which is given by

```
⟨assign|render-proof|
    ⟨macro|which|body|
        ⟨surround||⟨htab|0.5fn|0⟩⟨qed⟩|
            ⟨render-remark|which|
                body⟩⟩⟩⟩
```

The surround primitive provides a way to decorate large blocks of text (so-called "block markup") at the left and at the right. In this case, the proof text is decorated with the end-of-proof symbol □, which is produced using the macro qed. The symbol is preceded by a "horizontal tab" that flushes it to the right margin. It now suffices to change the definition of the qed macro to ■. The above proof then reads:

> **Continued proof of the master theorem.** We now conclude by applying the master lemma. ■

At this point, we succeeded in replacing the □ symbol by ■, but let us push our explorations a bit farther. The render-remark macro is another even more central rendering macro, which is also used for remarks, notes, examples, etc. Its source code is as follows:

```
⟨assign|render-remark|
    ⟨macro|which|body|
        ⟨render-enunciation|⟨remark-name|which⟨remark-sep⟩⟩|body⟩⟩⟩
```

As you can see here, render-remark depends on its turn on a yet more central macro render-enunciation, which is also used for theorems and exercises. This additional level of indirection is due to the fact that the bodies of major theorem-like enunciations are usually emphasized, whereas proofs, remarks, and other less prominent enunciations do not the change the font of the main body. Examination of the definition of render-remark also reveals the possibility to specify an alternative rendering for the names of remarks (e.g. REMARK 2.3) and the separator between this name and the corresponding body (usually a dot). Further investigations lead us to the source code of render-enunciation:

```
⟨assign|render-enunciation|
    ⟨macro|which|body|
        ⟨padded*|
            ⟨surround|which|⟨yes-indent*⟩|
                body⟩⟩⟩⟩
```

We have now reached the explicit graphical layout macro padded* that is used to add vertical whitespace before and after the enunciation.

```
⟨assign|padded*|
    ⟨macro|body|
        ⟨padded-normal|large-padding-above|large-padding-below|
            body⟩⟩⟩
```

The precise amount of spacing is controlled by the style parameters *large-padding-above* and *large-padding-below*. For all macros that depend on padded*, this leads to entries Focus ▸ Preferences ▸ Large padding above and Large padding below in the Style parameters group. If your cursor is inside a proof, then

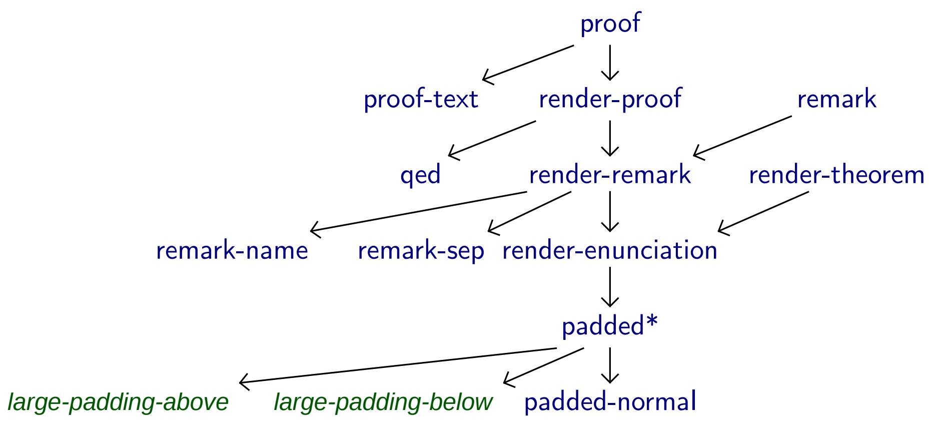


Figure 12.4. The dependency tree for the proof macro.

you may change these style parameters. The padded-normal macro is defined directly in terms of typesetting primitives that cannot be further customized:

```
⟨assign|padded-normal|
    ⟨macro|before|after|body|
        ⟨surround|⟨vspace*|before⟩⟨no-indent⟩|⟨htab|0fn|first⟩⟨vspace|after⟩|
            body⟩⟩⟩
```

Figure 12.4 summarizes the recursive dependencies of the proof macro. Once again: before customizing an existing macro defined by the selected style, it is recommended to investigate its dependency tree and carefully identify those submacros that control the behavior that you wish to change.

12.2 Pilgrimage to the source

In the previous section, we have seen that it can be useful to select the "source code" rendering when editing macros. You may actually select this rendering style for any $\mathrm{T_EX_{MACS}}$ document using ⌘⌥S or Document ▸ Source ▸ Edit source tree, and for any selected portion of text using ⌘-. For certain documents, this gives a better grip on their full structure and in particular reveals any hidden information such as folded content.

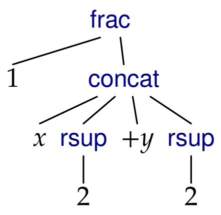
However, contrary to LaTeX and HTML (among others), the preferred representation of $\mathrm{T_EX_{MACS}}$ documents is *not* based on ASCII texts. This makes the concept of "source code" somewhat ill-defined for $\mathrm{T_EX_{MACS}}$ documents. As we will explain in this section, the best mental representation of a $\mathrm{T_EX_{MACS}}$ document is to regard it as an abstract tree. There are many ways to render such trees on a screen, some of which are closer to the printed end-result, and some of which expose more of the hidden internal structure.

12.2.1 Documents as trees

From a more conceptual perspective, TeXmacs systematically regards documents as *trees*. Consider for example the formula

$$\frac{1}{x^2+y^2}. \tag{12.2}$$

TeXmacs internally represents this formula by a tree[12.1]



(12.3)

The nodes of the tree are labeled by markup elements (internal primitives or user-defined macros) and the leaves of the tree are ordinary text strings.

Pursuing our conceptual perspective, one should distinguish between the abstract document tree and its graphical rendering. In particular, (12.3) is just an attempt to represent such a tree in a graphically pleasing way. The default rendering (12.2) is another representation and the "source code" rendering is yet another possible representation:

⟨frac|1|x⟨rsup|2⟩+y⟨rsup|2⟩⟩

The main virtue of "source code" with respect to other renderings is that it aims to make the full structure as transparent as possible. But even though TeXmacs proposes a default "source code" rendering, there is no intrinsic reason to prefer any particular such rendering over another one. For instance, scheme expressions provide an alternative that only involves plain ASCII text:

```
(frac "1" (concat "x" (rsup "2") "+y" (rsup "2")))
```

We will come back to this representation in chapter 14, since it is particularly useful when using SCHEME as an extension language for TeXmacs.

12.2.2 Selecting your preferred presentation of source code

The "source code" rendering of document trees can actually be customized via Document ▸ Source ▸ Preferences. We invite you to play around with the different possibilities in a document of your own (after enabling Document ▸ Source ▸ Source tree).

12.1. Notice that x^2 is represented as x⟨rsup|2⟩ and *not* as ⟨rsup|x|2⟩. This is more convenient for wysiwyg editing purposes, since you do not need to specify the precise mathematical expression that is being superscripted.

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 12.5. Different styles for rendering the same source tree.

First of all, you may choose your preferred "Main presentation style" among "angular", "scheme", "functional", and "LaTeX", as illustrated in the Figure 12.5.

Secondly, you may wish to reserve a special treatment for certain tags, such as the formatting tags concat and document for horizontal and vertical concatenation. In the menu Tags with a special rendering you may specify to which extent you want to treat such tags in a special way:

Raw. No tags receive a special treatment.

Format. All tags are rendered as source code, except for the formatting tags concat and document.

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

Maximal. At the moment, this option is not yet implemented. The intention is to allow you to write your own customizations and to allow for a special rendering of basic mathematical operations such as plus.

These different options are illustrated in Figure 12.6.

Raw

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

Format

```
⟨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 12.6. Different ways to render special tags.

None

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

Inline

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

Normal

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

All

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

Figure 12.7. Different levels of compactification.

Another thing that can be controlled by the user is whether the presentation of tags should be compact or stretched out across several lines. In general, so-called *inline tags* are more readable when represented in a compact way. Inline tags are used for small pieces of content in the middle of a paragraph, such as a reference or an inline formula. On the other hand, *block tags* are used for longer pieces of content that may span over several paragraphs, such as proofs or item lists. In source code rendering, they typically look better when they are stretched out over several lines. Note that certain tags (like em for emphasized text) can be used both as inline and as block tags.

The Compactification menu proposes several levels of compactification:

None. The tags are all stretched out across several lines.

Inline. 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.

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

The visual effect of the different levels of compactification is illustrated in Figure 12.7.

Finally, the Closing style menu allows you to specify the way closing tags should be rendered when the tag is stretched out across several lines. The rendering can either be minimalistic, compact, long, or recall the matching opening tag. The different options are illustrated in Figure 12.8.

Minimal

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

Compact

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

Long

```
⟨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 12.8. Different ways to render closing tags.

12.2.3 Data or program?

When designing your macros, you will quickly find out that TeXmacs macros have a dual character. Simple macros such as an abbreviation or a notation for diagonal matrices are usually visual in nature and best edited in the same way as ordinary text. More complex macros behave more like programs with local variables and subprograms. The source code representation tends to be more convenient for editing such macros. The borderline between textual data and macro programming is blurry: sometimes both aspects can be found in the same macro definition.

TeXmacs fully acknowledges this data-or-program duality and provides a few mechanisms to locally switch between normal and source style rendering. One typical use case is to quickly examine (and possibly modify) the "source code" of a small portion of text. For example, assume that we wish to study the markup that was used in order to produce "brown" and "jumps" in the fragment

> The quick brown fox jumps over the lazy dog.

Then it suffices to select "brown fox jumps" and press ⌘- (when activating the source tool using Tools ▸ Source macros tool, you may also use Source ▸ Activation ▸ Deactivate):

> The quick ⟨with|color|brown|brown⟩ fox ⟨move|jumps||0.5ex⟩
> over the lazy dog.

This technique inserts an additional (invisible) inactive* tag around the selection. You have to remove this tag in order to switch back to the original presentation, e.g. by placing your cursor after the "x" in "fox" and pressing ^⌫. Recall that the source code for the entire document can be edited using Document ▸ Source ▸ Edit source tree.

Conversely, when editing a macro definition in source mode, you may prefer the usual non-source rendering for the more graphical portions of text. Consider for instance the definition

```
⟨assign|new-icon|⟨macro|⟨icon|tm_new_x4.png⟩⟩⟩
```

You may prefer the following presentation that was obtained by selecting ⟨icon|`tm_new_x4.png`⟩ and pressing ⌘+:

```
⟨assign|new-icon|⟨macro|🗋⟩⟩
```

A more complex example is the following:

⟨assign|*diag*|⟨macro|*var*|*dim*|$\begin{pmatrix} \mathit{var}_1 & & \mathbf{0} \\ & \ddots & \\ \mathbf{0} & & \mathit{var}_{\mathit{dim}} \end{pmatrix}$⟩⟩

Here we activated the usual rendering for the matrix using ⌘+, but switched back to source code rendering for the arguments *var* and *dim* using ⌘-.

12.2.4 The ASCII religion

Let us briefly discuss one aspect of source code that may not be of direct practical interest, but which may give you some insight into the design philosophy behind TEXMACS.

Programmers are accustomed to the use of ASCII as their privileged representation of source code. Over decades, many tools have been developed in order to make ASCII-style programming highly efficient. However, it is not so clear that a line of C++ code such as

```
P = 23*pow(x,3)*pow(y,2)*z + 17*x*pow(z,4)
```
(12.4)

is easier to read than

$$P=23x^3y^2z+17xz^4 \tag{12.5}$$

Similarly, some masochism is required to prefer LATEX code

```
M = \left( \begin{array}{ccc}
  1 & \alpha_{1,2} & \alpha_{1,3}\\
  0 & 1 & \alpha_{2,3}\\
  0 & 0 & 1
\end{array} \right)
```
(12.6)

over

$$M=\begin{pmatrix} 1 & \alpha_{1,2} & \alpha_{1,3} \\ 0 & 1 & \alpha_{2,3} \\ 0 & 0 & 1 \end{pmatrix} \tag{12.7}$$

With an appropriate editor like TeXmacs, we do not only claim that (12.5) and (12.7) can be entered faster than (12.4) and (12.6): we may also regard them as a better way to represent "source code".

In fact, we regard it as an interesting challenge to develop a "source code editor" that is at least as efficient as traditional ASCII-based editors, but that allows for visually more attractive and readable presentations. The main advantage of existing editors is that the user is not constrained by the structure of a program. For instance, the following piece of code was made more readable through the manual insertion of spaces at appropriate places:

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

However, the precise formatting policy does not appear in the document, so manual intervention will be necessary if the variable `cond` is renamed `c`, or if the variable `holala` is renamed `hola`.

TeXmacs is not really intended to be a program editor yet, so no tools are provided for dealing with the above example in an automatic way. Nevertheless, the discussion applies in a lesser extent to the source code mode in TeXmacs. New users may feel somewhat constrained by the document structure and disoriented by the fact that ASCII-style editing habits not always apply. These disadvantages are still felt by more experienced users, but compensated by the benefits of the structured editing facilities. Nevertheless, it remains a challenge to make it even easier and more natural to edit source code.

12.3 Grouping your macros together

12.3.1 Preambles

The macro editor is an efficient tool for the definition of simple notations and for basic customizations of existing macros. The resulting macro (re-)definitions are automatically stored in the *preamble* of your document. More experienced TeXmacs users often do not use the macro editor and prefer to directly edit the preamble using ⌘⌥p or Document ▸ Part ▸ Show preamble. This has the advantage that you see all macro definitions at once. By putting them in a suitable order, you may also enhance their readability.

Preambles are edited much in the same way as ordinary documents, except that you are in source mode. Copying and pasting in particular continues to work as usual. This is a powerful technique for the definition of complex macros: first design a prototype macro application in an ordinary document, next copy and paste it into the body of your macro definition in the preamble, and finally perform the appropriate replacements of text by macro arguments.

Besides macros, you may also edit environment variables using the macro editor. Examples of environment variables are the page size, the left margin, the text color, or the number of the current subsection. Some of these environment variables can also be modified more directly using the Document menu. For example, the main text color for the document can be selected using Document ▸ Colors ▸ Foreground.

Just like the `\` key allows you to apply a macro whenever it is defined (see section 12.1.2), you may use it in order to obtain the value of any available environment variable. For example, `\` `c` `o` `l` `o` `r` `↩` yields the value of the current text color. Inside source code, TeXmacs uses a green color for such environment variables, as in *color*.

12.3.2 Style packages

The preamble contains style customizations that apply to one individual document. If you are particularly pleased with some of your customizations, then you may wish to bundle them in a *style package* that can be reused in other documents. One efficient way to start a new style package is Tools ▸ Macros ▸ Extract style package: this collects all existing customizations in your current document (both macros and environment variables) and puts them into a new style package.

Style packages are similar to usual TeXmacs documents, except that you edit them in source mode, just like preambles. When saving them to disk, you should also use the `.ts` extension instead of `.tm`. The native TeXmacs style packages are stored in the directory `$TEXMACS_PATH/packages`, where `$TEXMACS_PATH` points to the directory where you installed TeXmacs. You must put your own style packages in the same directory as the files that use them or in the directory `$TEXMACS_HOME_PATH/packages`, where `$TEXMACS_HOME_PATH` points to `~/.TeXmacs` on UNIX systems and to `C:\Documents and settings\`*`user_name`*`\Application Data\TeXmacs` on WINDOWS. After saving your style package in this way, you may apply it to any of your TeXmacs files using Document ▸ Style ▸ Add package ▸ Add other package.

Style files are a variant of style packages. Style files control the main document style, whereas style packages contain customizations that can be combined with other style files or packages. Style files must be complete in the sense that all major markup elements (such as section, theorem, equation, bibliography, strong, etc.) should be defined. Style packages typically contain extra personal notations or customizations of existing markup elements.

The complete style of your current document can be extracted using Tools ▸ Macros ▸ Extract style file. When saving style files to disk, you should again use the `.ts` extension. Native style files are stored in `$TEXMACS_PATH/styles`. You should put your own style files in `$TEXMACS_HOME_PATH/styles` or in the same directory as the files that use them. The main style is selected using Document ▸ Style and user-defined styles are automatically listed there.

12.4 The style-sheet language

So far, we have discussed how to introduce new notations and how to produce simple customizations of existing macros. For the design of more complex macros, $\mathrm{T_EX_{MACS}}$ provides a special *style-sheet language*. This language consists of a set of special primitives that allow you to locally modify an environment variable, choose between several renderings, perform length computations, etc. In this section, we give an introduction to some of the most important primitives. A full exposition is beyond the scope of this book; for more details, we refer to the reference guide that is found in Help ▸ Reference guide.

Throughout this section, we assume the editor to be in "source mode", so that a top-level Source menu should be available.

12.4.1 Assignments

All user-defined $\mathrm{T_EX_{MACS}}$ macros and style variables are stored in the "current typesetting environment". This environment associates tree values to string variables. 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 with primitive can also be used for local redefinitions of macros:

```
⟨with | strong | ⟨macro | body | ⟨with | font-series | bold | color | red | body⟩⟩ | ⟨strong |
strong⟩ text⟩
```

The value of an environment variable may be retrieved using the value primitive, as in the following code to increase a counter:

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

Here we recall that the default rendering of ⟨value | *my-counter*⟩ inside source code is *my-counter* (see sections 12.2.2 and 12.3.1). Our example actually shows that both left-hand sides of assignments and values of environment variables are emphasized in green, which should help to locate environment variables inside source code. Similarly, macro arguments can be recognized by their brown color.

We recall that the value of an existing environment variable can be retrieved using the `\` key. For instance, `\ m y - c o u n t e r ↩` yields *my-counter*. In source mode, you may also use the keyboard shortcuts `⌘^=`, `⌘^w`, and `⌘^v` to produce assign, with, and value tags.

12.4.2 Macro expansion

The main interest of the style-sheet language is the possibility to define macros. These come in three flavors: 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 ordinary macros.

Ordinary macros are 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. Notice that the default rendering of ⟨arg|*name*⟩ inside source code is *name*:

```
⟨assign|hello|⟨macro|name|Hello name, you look nice today!⟩⟩
```

It is possible to call a macro with fewer or more arguments than the expected number. Superfluous arguments are simply ignored. Missing arguments take the nullary uninit primitive (with rendering ?) 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!⟩⟩⟩
```

You are allowed to use macros as values:

```
⟨assign|my-macro-copy|my-macro⟩
```

However, when using this style of macro programming, one should keep in mind that TEXMACS macros use a call-by-name evaluation strategy, contrary to functional programming languages such as SCHEME (see section 12.4.4 below). The compound tag may be used to apply macros that are the result of a computation:

```
⟨assign|overloaded-hi|
    ⟨macro|name|
        ⟨compound|
            ⟨if|⟨nice-weather⟩|happy-hi|sad-hi⟩|
            name⟩⟩⟩
```

12.4.3 Formatting primitives

Most $\TeX_{\text{MACS}}$ presentation tags can be divided in two main categories: *inline* tags and *block* tags. Inline tags are used for small pieces of text, whereas block tags can contain text that spans over several paragraphs. For instance, frac is a typical inline tag, whereas theorem is a typical block tag. Some tags (such as strong) are inline if their argument is inline and block otherwise.

The most primitive inline tag concat is used for horizontal concatenation: ⟨concat|⟨with|*color*|blue|blue⟩|⟨em|emphasis⟩⟩ is rendered as blue*emphasis*. Similarly, the most primitive block tag document is used for vertical successions of paragraphs: ⟨document|First|Second|Third⟩ is rendered as

```
First

Second

Third
```

The concat and document tags are so common that their names are actually hidden for the default rendering of source code (see section 12.2.2 and Figure 12.6).

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. For example, ⟨concat|Left|⟨document|Top|Bottom⟩|Right⟩ yields

```
LeftTop    Right
     Bottom
```

instead of the expected

```
LeftTop

BottomRight
```

If you need to surround a block tag with inline text, then you must use 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.**" on the left and a "right-flush" on the right-hand side. Flushing to the right is important in order to make the blue bounding box around the theorem look nice when editing the body of the theorem.

In most cases, TeXmacs 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 be used both as an inline tag and as a block environment. When placing your cursor just before the with-tag and hitting ↩ followed by ⌫, you obtain

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

These actions inserted a document tag around the body of the macro. The document tag itself is invisible (you should select Tags with special rendering ▸ raw in Document ▸ Source ▸ Preferences to make it visible), but its presence is indicated through the stretched rendering. Since the body of the macro is now a block, your tag very-important automatically becomes a block environment as well.

Another important property of tags is whether they contain normal textual content or tabular content. For example, consider the definition of the standard eqnarray* tag (with a bit of the presentation markup being 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 extend 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 only specify the final presentation of documents on paper: they also determine the way documents are presented during the editing phase. Above, we have already mentioned the use of the right-flush tag in order to improve the rendering of visual border hints. Similarly, flags can be used to indicate the presence of invisible arguments while editing a document:

```
⟨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 that are removed when printing the document (see section 7.6.2).

12.4.4 Evaluation control

The Source ▸ Evaluation menu contains several primitives to control the way expressions in the style-sheet language are evaluated. These primitives may in particular be used for the definition of "meta-macros" whose purpose is to define or redefine other macros. One typical example is the new-theorem meta-macro for the definition of new theorems:

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

When expanding ⟨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 such as SCHEME. We recall that SCHEME is the extension language for TeXmacs; see chapter 14 for a short introduction. The subtle differences between the style-sheet language and SCHEME are motivated by our objective to make it as easy as possible for the user to write macros for typesetting purposes. Assuming that you know the basics of SCHEME, it is instructive to examine the differences on a few examples.

When TeXmacs calls a macro ⟨macro|x_1|⋯|x_n|body⟩ with arguments $y_1, \ldots,$ y_n, the argument variables $x_1, \ldots, x_n$ are bound to the unevaluated expressions $y_1, \ldots, y_n$, and the body is evaluated with these bindings. The evaluation of y_i takes place each time the argument x_i is actually used during the evaluation of the macro. 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 literal bodies of SCHEME macros are evaluated twice, whereas the arguments of functions are evaluated only once. 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⟩⟩⟩
```

corresponds 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 or 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.

12.4.5 Control flow

The Source ▸ Flow control menu contains TeXmacs analogues of basic control flow primitives that can be found in conventional programming languages: if, case, while, and for-each. However, be warned that the conditional constructs are quite fragile: they only apply to inline content and it is recommended that the various alternatives only affect the rendering, not the structure. For instance, consider the fragment

```
⟨assign|weird|
    ⟨macro|body|
        ⟨if|⟨my-condition⟩|Hi body!|⟨label|body⟩⟩⟩⟩
```

In the "then part" of the if primitive, the *body* argument is a regular macro argument that can be edited by the user (we say that it is *accessible*). In the "else part", the *body* argument is a hidden parameter that should be deactivated before you can edit it. To remain on the safe side, $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ considers the *body* argument of the weird macro to be inaccessible, even when it is rendered according to the "then part".

The if primitive for conditional typesetting can be entered using ⌘^? and is by far the most important control flow primitive. We recommend to mainly use it in combination with the compound primitive, as follows:

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

In this example, rich-appendix is a block environment consisting of a title and a body, and that 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|rich-appendix|
    ⟨macro|title|body|
        ⟨if|
            ⟨long-document⟩|
            ⟨chapter-appendix|title|body⟩|
            ⟨section-appendix|title|body⟩⟩⟩⟩
```

The if primitive can 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, $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ is not clever enough to detect which arguments are optional and which arguments are accessible. This information should be specified manually using the drd-props primitive; see the reference guide for more information.

12.4.6 Computational markup

The menus Source ▸ Arithmetic, Text, Tuple, and Condition contain 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 red variant:

```
⟨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. Precise positioning is possible through appropriate computations with lengths:

```
⟨assign|center-axis|⟨macro|body|⟨move|body||⟨minus|0.5ex|0.5h⟩⟩⟩⟩
```

The purpose of this macro is to vertically center the argument at the position of the fraction bar. This is achieved by moving it up by half the height `0.5ex` of an x-character and then back down by half the height `0.5h` of the argument itself (the length unit `h` is automatically defined to be the height of the main body inside the move primitive).

Chapter 13
Designing with style

The previous chapter gave a first introduction to the stylesheet language and how to add your own macros. We are now ready to dive deeper into this topic and investigate how to design complete document styles and style packages.

Scientific documents use a variety of style elements: section titles, item lists, theorems, headers, etc. New document styles are meant to offer the complete functionality of the built-in styles, but with a special design, which might for instance be dictated by the "graphical charter" of a journal. Since it would be cumbersome to redevelop all standard macros from scratch, new styles are typically designed as customizations of existing ones. The bulk of this chapter is dedicated to explaining how this works.

Style packages correspond to additional customizations that can be combined with any base style. This is typically useful for personal notations or when one or more macros need to be adapted in order to achieve a certain graphical effect. The explanations in this chapter may serve for the development of style packages that customize the standard styles.

Advanced users may also design style packages for new types of functionality such as literate programming or interactive courses. Such style packages are usually accompanied by extensions to the editor itself, in the form of keyboard shortcuts, special menu entries, or more complex editing routines. The next chapter gives a first glimpse of how to develop such enhancements.

13.1 Some words of caution

We recall that one of the main aims of beautiful typesetting and graphical design is to be "invisible": readers should focus on content and not be distracted by typesetting details. This is a quite delicate task which is really a job for experts on typography. Entire books have been written on this topic; see for instance [4, 40].

With the advent of typesetting software, the bulk of the traditional design tasks reduces to the development of high quality document styles. *A priori*, this still requires the same kind of expertise, so why would you want to this yourself? One reason could be that you wish to reproduce the style of a particular journal. But maybe you just got bored aping your colleague next door and prefer to have some fun creating your own distinctive style instead.

If you go down this road and design your own style, then it is tempting to introduce all kinds of fancy decorations for standard environments, such as boxes around theorems, drop shadows for section titles, your favorite pictures for list items, etc. Keep in mind that such phantasies may quickly saturate and irritate your readers. Instead, we invite you to search for more subtle ways to distinguish yourself. This can often be achieved through the careful selection of a few fonts, or by modifying some spacings or separators. For example, the simple use of a sans serif font for theorems will already distinguish your style from all built-in TeXmacs styles:

Theorem 13.1. *This statement has a distinctive look & feel.*

Notice that it is not necessary to refrain from all forms of ornaments. For instance, it may be all right to use the following type of main section titles:

——— My fancy section ———

Although such titles are somewhat "aggressive", they tend to occur only a few times in a document. As long as this is the only fantasy in your document style, this remains acceptable, while immediately ensuring a distinctive look and feel.

Another thing that you should worry about is related to the dual "data versus program" nature of TeXmacs macros that we first mentioned in section 12.2.3. Now the programming side becomes more significant when macros are grouped together in style files or packages. In many respects, the development of style files is analogous to writing software libraries, with macros in the role of library functions and subroutines. Besides aesthetical considerations, it is therefore important to think about issues like dependencies, maintainability, and orthogonality (i.e. how well do your macros combine with macros from other packages?).

Once again, you should be guided by the notion of simplicity. The simpler your personal customizations are and the more they rely on standard mechanisms, the more likely they will continue to work well with future versions of TeXmacs and in combination with other style packages. For example, if your macros crucially depend on some obscure internal macro that you found in one of the TeXmacs style packages, then they will break down if the internal macro happens to disappear in a future version of TeXmacs. Similarly, your macros may fail to work in combination with alternate styles that do not implement the required internal macro. For similar reasons, you should avoid complex macros that adjust the typesetting process in subtle ways: ongoing progress in the TeXmacs typesetter may make it difficult to maintain such macros.

13.2 Anatomy of a style package

Before you start designing your own style files, we recommend you take a closer look at the existing style files. One fairly complete example is the `svjour` base style for articles published by SPRINGER VERLAG. This style is selected using Document ▸ Style ▸ Article ▸ Springer ▸ svjour. After selecting this style, you may open the corresponding source file using Document ▸ Style ▸ Edit style.

The first line of the `svjour` style file specifies the style packages on which the style is based. It essentially uses the same style packages as the standard `article` style, together with one additional package `std-latex`:

```
⟨use-package|std|env-base|env-math|env-float|env-program|header-article|
title-base|section-article|std-latex⟩
```

The `std-latex` package is useful when adapting a LaTeX style file to TeXmacs, since it provides several macros that allow you to specify global style parameters in "the LaTeX way". For instance, the `svjour` style uses the following lines to specify the global margins and text width—both for single and double column articles:

```
⟨assign|tex-odd-side-margin|⟨macro|⟨if|⟨equal|par-columns|1⟩|0pt|-30pt⟩⟩
⟨assign|tex-even-side-margin|⟨macro|⟨if|⟨equal|par-columns|1⟩|0pt|-30pt⟩⟩
⟨assign|tex-text-width|⟨macro|⟨if|⟨equal|par-columns|1⟩|25.5cc|17.8cm⟩⟩
```

Various other global style parameters are specified in a similar way.

The remainder of the `svjour` style file redefines some of the most significant macros that determine the layout of articles. One may distinguish the following main categories:

- Specification of the fonts and font sizes to be used.
- Rendering of sectional macros.
- Specification of headers and footers to be used.
- Theorem-like environments.
- Item lists and enumerations.
- Rendering of the document title and the abstract.

Let us briefly study how some of these customizations were carried out for `svjour`. More general and detailed explanations will be provided in the sections below.

Fonts and font sizes

Style files for journals often specify the main document font and its variants with care. The `svjour` style specifies the default font size to be `10pt` with a `0.2em` interline space, but also redefines the standard tags for other font sizes, such as small, large, smaller, etc. For example, the default rendering of footnotes uses a smaller font size that is specified through the smaller tag. The `svjour` style redefines it to use a `9pt` font with a `2pt` interline space:

```
⟨assign|smaller|⟨macro|x|⟨with|font-base-size|9|par-sep|2pt|x⟩⟩⟩
```

Sectional macros

After the main document font and page layout, the rendering of section titles is the next most significant characteristic of a scientific document style. Most journals use bold, emphasized, or small capital fonts, and carefully specify the amount of vertical space to be inserted before and after titles. Styles that were adapted from LaTeX by means of the `std-latex` package may reproduce TeX-style rubber spaces using the tex-len macro. For instance, `svjour` renders main section titles in the following way:

```
⟨assign|section-title|
    ⟨macro|name|
        ⟨sectional-normal-bold|
            ⟨concat|
                ⟨vspace*|⟨tex-len|21dd|4pt|4pt⟩⟩|
                ⟨normal-size|name⟩|
                ⟨vspace|⟨tex-len|10.5dd|4pt|4pt⟩⟩⟩⟩⟩⟩
```

Here sectional-normal-bold is a utility macro for rendering bold section titles. It also takes care of a few other issues such as forbidding page breaks just after the title.

Headers and footers

Most styles determine headers and footers as a function of the title and names of the authors. TeXmacs provides a few "call-back macros" for this purpose: header-title (document title), header-author (document author), header-primary (primary sections), and header-secondary (secondary sections). For example, the header-author macro is called with the name of the author as an argument, when specifying this information in the title of your document. The `svjour` style exploits this mechanism to redefine the even page header so as to contain the author's name:

```
⟨assign|header-author|
    ⟨macro|name|
        ⟨assign|page-even-header|
            ⟨small|
                ⟨no-indent⟩⟨page-number⟩⟨htab|5mm⟩name⟩⟩⟩⟩
```

Theorem-like environments

Scientific articles require many technical markup elements like theorems, enumerations, algorithms, technical pictures, etc. Full-fledged styles may redefine the rendering of such markup elements. The standard $\mathrm{T_EX_{MACS}}$ styles contain a large number of macros that allow you to customize specific rendering aspects. For instance, the main rendering of theorem-like environments is controlled through the render-theorem macro. But it is also possible to change the font for the text "THEOREM 3.6." using theorem-name, as well as the separator "." after the number using theorem-sep. For our example `svjour` style, we essentially have

```
⟨assign|theorem-name|⟨macro|name|⟨with|font-series|bold|name⟩⟩⟩
⟨assign|theorem-sep|⟨macro|. ⟩⟩
⟨assign|render-theorem|
    ⟨macro|which|body|
        ⟨padded-normal|1fn|1fn|
            ⟨surround|⟨theorem-name|which⟨theorem-sep⟩⟩||
                ⟨with|font-shape|italic|body⟩⟩⟩⟩⟩
```

In the definition of render-theorem, the padded-normal macro is used for specifying the vertical padding around theorem-like environments. The `std-utils` style package contains various other handy utility macros of this kind.

Lists and enumerations

Different nesting levels of list environments typically require separate rendering styles. $\mathrm{T_EX_{MACS}}$ provides the macros enumerate-1, enumerate-2, etc. for top-level enumerations, second level sub-enumerations, and so on. Most style files limit themselves to three nesting levels; for `svjour`, we have

```
⟨assign|aligned-accolade-item|⟨macro|x|⟨aligned-item|⟨with|font-shape|right|
{x}⟩⟩⟩⟩
⟨new-list|enumerate-1|aligned-dot-item|identity⟩
⟨new-list|enumerate-2|aligned-accolade-item|⟨macro|x|⟨number|x|alpha⟩⟩⟩
⟨new-list|enumerate-3|aligned-dot-item|⟨macro|x|⟨number|x|roman⟩⟩⟩
```

The new-list construct is used to define the macros enumerate-1, enumerate-2, and enumerate-3; its second argument is a macro that is used for rendering items; the third argument contains a macro that is used for transforming the numbers of items. The macros aligned-item and aligned-dot-item are standard rendering macros that can be redefined. The rendering of lists themselves is controlled through the render-list macro.

Document titles and abstracts

One of the most complex parts of style files concerns document titles and abstracts. The difficulty stems from the fact that title and abstract information should really be thought of as a collection of metadata for the document. The rendering then proceeds in two steps: one first has to collect and reorganize the data in an appropriate way and then call lower-level rendering macros in an appropriate order. Here one has to keep in mind that a lot of information (keywords, subject classifiers, affiliations, etc.) is optional or may need to be recombined (various authors with multiple affiliations).

It is recommended that style files only customize the final rendering macros for title and abstract information. The title mainly consists of a succession of "blocks" that are rendered using the doc-title-block macro. Specific kinds of information are rendered using dedicated macros doc-render-title, doc-subtitle, doc-date, etc. For example, the rendering of the main document title is defined as follows for the `svjour` style:

```
⟨assign|doc-render-title|
    ⟨macro|x|
        ⟨surround||⟨vspace|11.24pt⟩|
            ⟨doc-title-block|⟨larger|⟨with|math-font-series|bold|font-series|bold|
            x⟩⟩⟩⟩⟩⟩
```

13.3 Some tips before things get technical

Before going deeper into the technical details about how to control specific style elements, let us first discuss a few general tips for the development of style packages.

Which macros to redefine

Most standard style files and packages consist of a succession of macro definitions and specifications of default values for environment variables. When customizing an existing style, it is important to carefully select the macros and environment variables to be redefined.

Now certain tags in the standard style files are directly exposed to the end-user through the interface. For instance, you have direct access to the tags strong and section through the menu entries Insert ▸ Content tag ▸ Strong and Insert ▸ Section ▸ Section. Other tags such as section-title or render-theorem are rather provided for customization purposes by developers of style files. Some style packages also define auxiliary helper macros for internal purposes.

Although sufficiently simple tags like strong can be redefined directly in your own style files, this is not recommended for more complex tags such as section. Indeed, the section tag takes care of many tasks: rendering the title, resetting the counter for subsections, entering the title into the table of contents, and so on. In order to customize complex tags of this kind, you should rather redefine one or more of the "companion tags" section-title, section-clean, section-toc, etc. that control these more specific subtasks.

Customizing existing macros

Imagine that you want to change the rendering of a given tag, like lemma. We have just seen that TeXmacs provides a set of carefully designed companion macros that can be customized to modify specific aspects of the rendering. For instance (see section 13.2), you are supposed to redefine 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 companion 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. Besides, you may not want to spend too much time on understanding the macro hierarchy of TeXmacs, thereby ignoring the very 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 the lemma macro to a safe place and redefine it 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⟩⟩⟩⟩
```

However, be warned that this mechanism is somewhat fragile: if the name orig-lemma was already in use (for instance, if you already performed a similar customization in another style package), then you may introduce a circular dependency lemma → orig-lemma → lemma → orig-lemma → ⋯. Here we note that TeXmacs contains no safeguards against programming errors by developers of style packages: infinite loops of the above type will simply crash the editor.

One obvious fix is to choose the backup name orig-lemma with more care; e.g. uncolored-lemma might be more appropriate. It is also a good idea to define the backup macro if only if no macro with the same name already exists; this can be done by using the provide primitive instead of assign:

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

At the end of section 13.7, the redefinition of inc-theorem shows a more robust but subtle mechanism for customizing existing macros.

Local customizations

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 $\mathrm{T_EX_{MACS}}$, you may simulate this behavior by redefining macros inside a with. For example, imagine that we want to suppress the inter-paragraph space inside lists within theorem-like environments. 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|⟨orig-render-list|⟨with|par-par-sep|0fn|x⟩⟩⟩|
                ⟨orig-render-theorem|name|body⟩⟩⟩⟩⟩
```

On the one hand, 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.

Rely on the "standard utilities"

The style package `std-utils` contains various useful "helper macros" that should make it easier to develop style packages. It mainly contains macros for

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

It is good practice to rely on these standard macros whenever possible. Indeed, the low-level $\mathrm{T_EX_{MACS}}$ 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.

Internationalization

Certain tags like theorems, figures, or bibliographies need to print English text like "Theorem", "Figure", or "References". In order to allow this text to be customized, such tags usually come with companion macros theorem-text, figure-text, bibliography-text, etc. For example, the `svjour` style redefines the figure-text macro to print "Fig." instead of "Figure". In order to automatically translate the text from English into the current language, you should pass it as an argument to the localize macro. For instance, the `section-base` package contains the following default definition of bibliography-text:

```
⟨assign|bibliography-text|⟨macro|⟨localize|Bibliography⟩⟩⟩
```

The directory `$TEXMACS_PATH/langs/natural/dic` contains the dictionaries for translations into the supported languages. Here `$TEXMACS_PATH` is the place where TeXmacs is installed on your system.

Converting LaTeX styles

Style files for TeX/LaTeX are particularly complex and ill-behaved, with lots of auxiliary macros and style parameters without clear semantics. For this reason, the built-in LaTeX converters perform rather poorly on style files. TeXmacs provides equivalents for some of the most important LaTeX styles. If you wish to mimic another style, then we recommend that you proceed as follows:

- Try to import the LaTeX style file. Most of the structure will be lost, but the converter sometimes manages to import at least a few macros and environment variables. The result of the conversion may therefore provide a reasonable start for the development of a TeXmacs equivalent for your LaTeX style.

- Determine an existing TeXmacs style that comes as close as possible to the desired style and base your new style on it using use-package. Notice that the converter from the previous step may already have come up with a good proposal.

- Add a line ⟨use-package|std-latex⟩ to your style file and customize the utility macros from that package in order to specify the most important layout parameters.

- Patiently customize various other macros in order to match the desired style, following the mechanisms that will be described in the subsections below.

Concerning the third step, we notice that TeXmacs does not always use the same conventions as TeX/LaTeX when it comes to global layout parameters. For example, the style parameters `\oddsidemargin` and `\evensidemargin` for left margins on odd and even pages do not mean what you would think, since TeX automatically adds one extra inch. The `std-latex` package provides macros tex-odd-side-margin and tex-even-side-margin that mimic this somewhat twisted behavior in TeXmacs. Various other style parameters `\textwidth`, `\topmargin`, `\jot`, `\abovedisplayskip`, etc. admit similar analogues tex-text-width, tex-top-margin, tex-jot, tex-above-display-skip.

The `std-latex` package also defines a macro tex-len with three arguments `default-length`, `plus`, and `minus` that emulates the syntax of TEX rubber lengths. For example, ⟨tex-len|1 em|0.5 em|0.25 em⟩ stands for the rubber length with default, minimal, and maximal values 1 em, 0.75 em, and 1.5 em, respectively. The corresponding TEXMACS syntax for this rubber length is ⟨tmlen | 0.75 em|1 em|1.5 em⟩. See pages 62 and 7.1 for more information on TEXMACS lengths.

13.4 Customizing the general page layout

The general layout of a document is mainly modified by setting the appropriate environment variables for page layout and paragraph layout (see the reference manual for a complete list of these variables). For example, by including the following lines in your style file, you can set the page size to `letter` and the left and right margins to 1.25 in:

```
⟨assign|page-type|letter⟩
⟨assign|page-odd|1.25in⟩
⟨assign|page-even|1.25in⟩
⟨assign|page-right|1.25in⟩
```

Recall from section 3.10 that the margins may be different on even and odd pages; the environment variables *page-odd* and *page-right* correspond to the left and right margins on odd pages.

Note 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 of TEX/LATEX. Typically, this allows you specify the global layout using declarations such as

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

Notice that *page-odd*, *page-even*, etc. are lengths, whereas tex-odd-side-margin, tex-even-side-margin, etc. are macros that return a length.

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 get called when the document title or author are specified, or when a new primary or secondary section is started (by default, primary sections correspond to chapters in books, and to 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|
                    ⟨page-the-page⟩⟨htab|5mm⟩⟨unquote|title⟩⟩⟩⟩⟩⟩
```

13.5 Processing title information

The "titles" of scientific documents usually carry a lot of additional information about the authors, their affiliations, the creation date, grant acknowledgments, etc. For this reason, $\mathrm{T_EX_{MACS}}$ treats the title information as a miniature database, and the graphical rendering proceeds in two phases: the most relevant information is first extracted from the database and reorganized. The actual rendering is done at a second stage, using dedicated macros for this purpose.

The first stage is fairly complex, since one has to deal with various optional data fields and potentially multiple authors with multiple affiliations. Various styles may present the data in different orders and one has to decide how to present common affiliations among coauthors (see section 3.3.1 for a few common styles). Since the required rewritings are rather intricate, they are not performed by $\mathrm{T_EX_{MACS}}$ macros, but rather through "external" SCHEME routines whose precise description falls beyond the scope of this book.

The second stage is more straightforward: for each kind of title information, there is a corresponding rendering macro that can be customized by particular styles. For instance, the main title, an optional subtitle, the creation date, and miscellaneous extra title fields are rendered using the macros doc-title, doc-subtitle, doc-date, and doc-misc. So if the date should appear in a bold italic typeface and at a distance of at least `0.5fn` from the other title fields, then you may redefine doc-date as

```
⟨assign|doc-date|
    ⟨macro|x|
        ⟨surround|⟨vspace*|0.5fn⟩|⟨vspace|0.5fn⟩|
            ⟨doc-title-block|⟨with|font-shape|italic|font-series|bold|x⟩⟩⟩⟩⟩
```

The helper macro doc-title-block should be used for rendering atomic blocks of title information; many styles implement this macro by centering title blocks, whereas other styles rather align them to the left. $\mathrm{T_EX_{MACS}}$ also uses the macro doc-make-title for encapsulating all title information. You may specify an amount of padding between titles and the main text by customizing this macro.

The rendering of author information is done using similar macros author-name, author-affiliation, author-email, author-homepage, and author-misc. These macros should rely on the macro doc-author-block for rendering atomic blocks of author information. In addition, TeXmacs provides variants author-affiliation-note, author-email-note, author-homepage-note, and author-misc-note that are used when several authors share common information. These variants take one additional argument that contains the symbol (such as †) that is used to link the shared information to the corresponding authors.

Author information is often rendered differently for documents with one versus several authors. In case of a single author, the doc-author macro is used for rendering the block with all author information. This macro behaves similarly to doc-title, doc-subtitle, etc., and should rely on doc-title-block for its rendering. If there are multiple authors, then the doc-authors macro is used for rendering the complete list of authors (the macro uses one argument for each author), whereas the information of each individual author is encapsulated inside a block that is rendered using the doc-authors-block macro.

Some further global metadata are provided in the abstract rather than in the title. The rendering of the abstract is controlled via the macro render-abstract. In addition, TeXmacs provides the macro abstract-keywords for rendering a list of keywords (one argument for every keyword) and the macros abstract-acm, abstract-msc, and abstract-pacs for specifying ACM, AMS, and PACS subject classifiers.

13.6 Customizing sectional tags

TeXmacs provides the same sectional tags as LaTeX: part, chapter, section, subsection, subsubsection, paragraph, subparagraph, and 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.

One important additional "predicate macro" is sectional-short-style. If 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 books use the long style.

The rendering of a sectional tag *x* is controlled through the macros *x*-sep, *x*-post-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 example, after redefining

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

sectional titles typically look as follows:

13.1 – Hairy GNUs

Similarly, *x*-post-sep prints the separator between the section title and subsequent text. 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 sectional-normal macro comes from the `section-base` style package. You may find several similar macros sectional-normal-italic, sectional-centered-bold, etc. there for some of the most frequent ways to render section titles.

There are two main rendering styles for sectional titles. By default, paragraphs and subparagraphs use a "short" rendering style, with a body that starts immediately after the title:

My paragraph Blah, blah, and more blahs…

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

My section

Blah, blah, and more blahs…

When customizing the rendering of sectional titles, we recommend that you follow the same conventions: render paragraphs and subparagraphs in the short way and all others in the long way.

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

- The call-back macro *x*-clean can be used for resetting some counters when a new section is started. For example, 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.

13.7 Customizing numbered textual environments

TEX$_{\text{MACS}}$ provides three standard types of numbered textual environments: theorem-like environments, remark-like environments, and exercise-like environments. The first two types of environments are also called "enunciations". The following aspects of numbered textual environments can easily be customized:

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

Defining new environments

It is possible to introduce new environments 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 example, you may wish to define the environment experiment by

```
⟨new-theorem|experiment|Experiment⟩
```

The text "Experiment" will automatically be translated if your document is written in a foreign language, provided that there is an entry for this word in the TEX$_{\text{MACS}}$ dictionaries (see the section on internationalization on page 231).

Customizing the rendering

The main rendering of numbered textual environments can be customized by redefining the macros render-enunciation, render-theorem, render-remark, and render-exercise. 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, 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|italic|par-left|⟨plus|par-left|1.5fn⟩|body⟩⟩⟩⟩⟩
```

This redefinition produces the following effect:

THEOREM 13.2. *This is a theorem that has been indented.*

The macros render-theorem and render-remark are both based on the macro render-enunciation. The only difference between theorems and remarks is that theorems typically use an italic font. Note that proofs are rendered using render-proof; this macro is based on render-remark.

As you may have noticed by examining the above redefinition of render-theorem, it is also possible to customize the way names of theorems are printed or redefine the separator between the name and the body. Indeed, these aspects are controlled by the macros theorem-name and theorem-sep. For example, consider the following redefinitions:

```
⟨assign|theorem-name|⟨macro|name|⟨with|color|dark red|font-series|bold|font-shape|small-caps|name⟩⟩⟩
⟨assign|theorem-sep|⟨macro|: ⟩⟩
```

Then theorem-like environments will be rendered as follows:

PROPOSITION 13.3: *This proposition is rendered in is a fancy way.*

Customization of the numbering

All numbered environments such as sections and theorems come with counters *section-nr*, *theorem-nr*, etc. In order to increase, reset, or display the counter *theorem-nr*, you should use the companion macros inc-theorem, reset-theorem, and display-theorem. By redefining these macros, you may customize the way in which environments are numbered. 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 any additional action that might have been undertaken by the previous value of the macro inc-theorem. Indeed, ⟨unquote | *inc-theorem*⟩ gets replaced by the previous value of the macro inc-theorem. This macro has zero arguments and ⟨compound|⟨*unquote*|*inc-theorem*⟩⟩ simply evaluates its body.

TeXmacs organizes counters in various counter groups, which allows you to simultaneously reset all counters of a certain type at the start of a new section. The following code from the `number-long-article` style package is used in order to prefix all standard environments (theorems, equations, figures, etc.) 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⟩⟩
```

13.8 Customizing list environments

Item lists and enumerations are made up of two main ingredients: the outer list environment and the inner list items. For instance, consider the following list, together with its corresponding source code:

```
⟨itemize|
    ⟨item⟩First item.
    ⟨item⟩Second item.
    ⟨item⟩Third item.⟩
```

- First item.
- Second item.
- Third item.

Then the itemize tag corresponds to the outer list environment, whereas the bullets of the inner list items are produced by the item tag.

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 example, 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.) and reduces their right margin by `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 nested 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. Both these macros take the text for marking the item as their only argument (e.g. a bullet or a number). The macro aligned-item aligns the marker to the right (so that subsequent text is left-aligned), whereas compact-item aligns it to the left (so that subsequent text may not be aligned).

For example, consider the following redefinition of aligned-item:

```
⟨assign|aligned-item|
    ⟨macro|x|
        ⟨concat|
            ⟨vspace*|0.5fn⟩|
            ⟨with|par-first|-3fn|⟨yes-indent⟩⟩|
            ⟨resize|⟨embbb|x⟩|⟨minus|1r|2.5fn⟩||⟨plus|1r|0.5fn⟩|⟩⟩⟩⟩
```

Then items inside all list environments with aligned items will be "blackboard emboldened":

- Items in aligned description lists are rendered using aligned-item.

  C1. First condition.

  C2. Second condition.

- Items in compact description lists are rendered using compact-item.

  Gnus and gnats. Nice beasts.

  Micros and softies. Evil beings.

In this example, we used ⟨resize|⟨embbb|x⟩|⟨minus|1r|2.5fn⟩||⟨plus|1r|0.5fn⟩|⟩ to put ⟨embbb|x⟩ inside a box of width `3fn`: the new left limit of the box is `2.5fn` to the left of the original right limit `1r`; the new right limit is `0.5fn` farther to the right. We next used ⟨with|*par-first*|-3fn|⟨yes-indent⟩⟩ to move the resulting box `3fn` to the left, thereby aligning it to the right[13.1].

Remark 13.4. Both aligned-item and compact-item are inline macros. They are also base macros for various other internal macros aligned-space-item, long-compact-strong-dot-item, etc. that are used for the rendering of the different types of lists (itemize-arrow, description-long, etc.).

The `std-list` package 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 that 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⟩⟩⟩
```

13.1. There are simpler ways to achieve the same effect, but the way we presented here has the advantage that the alignment remains correct when boxes are moved during line breaking.

Chapter 14
Extend beyond imagination

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

Why SCHEME? The choice may indeed seem somewhat odd if you are accustomed to more conventional programming languages such as C++, JAVA, or PYTHON. In particular, SCHEME's heavily parenthesized syntax may scare more than one person. But it turns out that this baroque syntax has several advantages for our application. For one thing, it is easy to learn. More importantly, it enables us to treat programs and data in a uniform manner, which is particularly useful in the light of section 12.2.3. Other major advantages of SCHEME are its interactivity and high level of abstraction.

This chapter starts with a crash course on SCHEME [56]. There are several more extensive books on this topic, such as [15, 11, 52]. We also recommend to take a look at the GUILE reference manual [22], since this is the SCHEME implementation on which TeXmacs is currently based.

The rest of the chapter provides an introduction to programming TeXmacs extensions in SCHEME. More explanations on this subject can be found in Help ▸ Scheme extensions. It is also instructive to take a look at those TeXmacs source files that are written in SCHEME. These files can be found in the `progs` subdirectory of the path `$TEXMACS_PATH` where you installed TeXmacs. For instance, `$TEXMACS_PATH/progs/math/math-kbd.scm` lists the keyboard shortcuts in math mode.

14.1 A quick introduction to SCHEME

14.1.1 SCHEME sessions and atomic data types

TeXmacs uses SCHEME through an interactive interpreter. For a first encounter with the language, we invite you to launch an interactive SCHEME session using Insert ▸ Session ▸ Scheme. This allows you to enter commands after the SCHEME prompt and execute them by pressing ↩:

```
Scheme] (+ 1 1)
2
```

As you can see in this stunning example, applying a function `f` to the arguments $a_1, \ldots, a_n$ is done using `(f a₁ ... aₙ)`. Besides numbers, other fundamental atomic data types in SCHEME are booleans (`#t` and `#f`), characters (`#\a`, `#\b`,

`#\c`,...), strings (`"hi"`, `"there"`,...) and symbols (`x`, `fun`, `gnu2018`, `my-sym`, `x/3*`,...). The following mini-session shows a few typical computations with such objects:

```
Scheme] (> 3 2)
#t
Scheme] (if (> 2 3) "yes" "no")
"no"
Scheme] (string->list "Bonjour")
(#\B #\o #\n #\j #\o #\u #\r)
Scheme] (string-append "Hi" " " "there" "!")
"Hi there!"
Scheme] (string->symbol "hop@boeh!stuff*")
hop@boeh!stuff*
Scheme] (symbol? 'my-sym)
#t
```

One difference between strings and symbols is the notation, since strings need to be double quoted. Moreover, strings are literal constants, just like the numbers `2` and `3.14`, whereas symbols are evaluated by SCHEME. One may prevent a SCHEME expression from being evaluated using `'` as a prefix operator; this explains why we used `'my-sym` in order to obtain the symbol `my-sym`. From a computational point of view, symbols are sometimes more efficient: the equality of two symbols can be checked in unit time, whereas all characters in two strings need to be compared.

Symbols are also used as names for variables and functions. We note that SCHEME is very flexible when it comes to names of functions: all characters can be used except for whitespace and the special characters `()[]{}#.,;'"‘\`. This makes it possible to use suggestive names `string->symbol` and `symbol?` for converters and predicates. Also note that the dash `-` tends to be used (instead of `_`) as a connector for multiple-word function names such as `string-append`. In particular, many function names are prefixed by the type of their principal argument.

14.1.2 Lists and SCHEME trees

Expressions of the form (x_1 ... x_n) are also called *lists* and they are one of the most fundamental data structures in SCHEME. Lists are either constructed using `list` (from its list of entries) or `cons` (from the first *head* entry and the *tail* list of remaining entries):

```
Scheme] (list (+ 1 2 3) "hopsa" (= 2 3.4))
(6 "hopsa" #f)
```

```
Scheme] (cons (* 1 2 3 4 5) (list 6 7 8 9 10))
(120 6 7 8 9 10)
```

Notice that the second method corresponds to the way lists are stored in memory: the empty list corresponds to a special constant, whereas non-empty lists are stored as pairs with the first element of the list and a pointer to the tail. One may test for emptiness using the `null?` predicate and access the head and the tail using the `car` and `cdr` accessors. Moreover, `(caar l)`, `(cadr l)`, etc. can be used as shorthands for `(car (car l))`, `(car (cdr l))`, etc.

```
Scheme] (define l (list 1 2 3))
Scheme] (list (null? l) (car l) (cdr l) (caddr l)
              (null? (cdddr l)))
(#f 1 (2 3) 3 #t)
```

Lists are also quite convenient for the representation of TeXmacs document fragments. For instance, `(frac (sqrt "a") "b")` represents the fraction $\frac{\sqrt{a}}{b}$. One may use the prefix operator `'` to construct explicit SCHEME expressions of this kind, as we saw before in the case of symbols. The general mechanism is called *quoting* and it is extremely convenient:

```
Scheme] (sqrt "a") ;; try to apply the function sqrt to "a"
Wrong type (expecting real number): "a"
Scheme] '(sqrt "a") ;; prevent evaluation using quoting
(sqrt "a")
```

SCHEME also offers the prefix operator `` ` `` for *quasi-quoting*. Like quoting, quasi-quoting switches off evaluation; however, the evaluation of specific parts of the expression can be switched on again using the prefix operators `,` and `,@` (called *unquoting* and *unquote-splicing*):

```
Scheme] `(hop (hola ,(+ 1 2) ,(string->list "hop"))
              (hola ,(* 3 4) ,@(string->list "hop")))
(hop (hola 3 (#\h #\o #\p)) (hola 12 #\h #\o #\p))
```

Those SCHEME expressions that correspond to snippets of TeXmacs documents will be called SCHEME *trees*. A SCHEME tree is either a string or a list of the form `(l` t_1 `...` t_n`)`, where `l` is a TeXmacs primitive or macro and $t_1,\ldots,t_n$ are other SCHEME trees. One may convert such SCHEME trees to actual TeXmacs trees and *vice versa* using the routines `stree->tree` and `tree->stree`:

```
Scheme] (stree->tree '(math (frac (sqrt "a") "b")))
```

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

```
Scheme] (define my-name "John")
```

```
Scheme] (stree->tree
         `(concat "My name is " (strong ,my-name) "."))
```

My name is **John**.

14.1.3 Declaring functions and macros

New SCHEME functions can be declared using `define`:

```
Scheme] (define (fibonacci n)
          (if (<= n 1) 1
              (+ (fibonacci (- n 1))
                 (fibonacci (- n 2)))))
Scheme] (fibonacci 25)
121393
Scheme] (define (.. i j)
          (if (< i j) (cons i (.. (+ i 1) j)) (list)))
Scheme] (.. 0 15)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14)
```

One important feature of SCHEME is that it is a functional programming language: functions can both be used as arguments and return values for other functions. For example, you may use `apply` and `map` to apply a function to a list of arguments or to each element of a list:

```
Scheme] (apply + (list 1 2 3 4))
10
Scheme] (map fibonacci (.. 0 15))
(1 1 2 3 5 8 13 21 34 55 89 144 233 377 610)
```

The following syntax allows use to define functions that return functions:

```
Scheme] (define ((unary-compose f g) x) (f (g x)))
Scheme] (unary-compose fibonacci fibonacci)
#<procedure #f (x)>
Scheme] ((unary-compose fibonacci fibonacci) 5)
34
```

SCHEME also provides syntactic sugar to declare functions with an arbitrary number of arguments. The following improvement of `unary-compose` shows an example:

```
Scheme] (define ((compose f . gs) . xs)
          (apply f (map (lambda (g) (apply g xs)) gs)))
Scheme] ((compose - * +) 1 2 3 4)
          ;; (- (* 1 2 3 4) (+ 1 2 3 4)
14
```

Notice that `(lambda (g) (apply g xs))` stands for the function that takes g as input and that returns `(apply g xs)`.

Yet another powerful feature of SCHEME is the possibility to enrich the language with new primitives, through the declaration of SCHEME *macros*. This is done using the keyword `define-macro` that has a similar syntax as `define`, but with the difference that the return value is evaluated a second time.

```
Scheme] (define-macro (sum i vals . body)
          `(apply + (map (lambda (,i) ,@body) ,vals)))
Scheme] (sum k (.. 0 10) (* k k))
          ;; (apply + (map (lambda (k) (* k k)) (.. 0 10)))
285
```

This example also illustrates the power of quasi-quotation in combination with the macro system.

TEXMACS heavily relies on this possibility to extend the SCHEME language with new macros. For instance, new keyboard shortcuts and menus can be defined using the `kbd-map` and `tm-menu` macros, by means of a convenient syntax. In fact, function and macro declarations themselves should be done using `tm-define` and `tm-define-macro`: as will be detailed in section 14.3.2 below, these enhanced versions of `define` and `define-macro` make it possible to overload functions and macros in a contextual way; they also allow you to enrich functions with extra information that may be exploited by the graphical user interface, such as "suggested values" for certain function arguments.

14.1.4 Basic control structures

Local variables in SCHEME are introduced using the keyword `let*`, whereas `set!` can be used to assign a new value to such a variable:

```
Scheme] (define (foo x)
          (let* ((a (* x x))
                 (b (+ a x)))
            (set! a (* a b))
            (+ a x)))
Scheme] (foo 100)
101000100
```

The `if` primitive can be used both as an instruction and as an expression that returns a value. SCHEME programmers also frequently rely on the `cond` primitive which provides an elegant abbreviation for expressions like

```
(if c1 expr1
    (if c2 expr2
        ...))
```

For instance:

```
Scheme] (define (mix l1 l2)
          (cond ((null? l1) l2)
                ((null? l2) l1)
                (else `(,(car l1) ,(car l2)
                        ,@(mix (cdr l1) (cdr l2))))))
Scheme] (mix (list 'a 'b 'c) (list 1 2 3 4 5))

(a 1 b 2 c 3 4 5)
```

The boolean combinators `and` and `or` can take an arbitrary number of arguments. In fact, the types of the arguments are allowed to be arbitrary with the property that any value distinct from `#f` is considered to be "true". More precisely, the evaluation of `(and` x_1 `...` x_n`)` stops as soon as the first x_i in the list evaluates to `#f` (in which the return value is `#f`). If this never happens, then the evaluation of x_n is returned. Similar rules apply for the `or` primitive and various other functions that manipulate boolean values.

```
Scheme] (define a #f)
Scheme] (define (set!a b) (set! a b) a)
Scheme] (list (and "hi" "there")
              (and "hi" (set!a 10) #f (set!a 100))
              a)

("there" #f 10)
```

14.2 Customizing the graphical user interface

14.2.1 Your personal boot file

So far, we have seen how to execute SCHEME commands from interactive sessions. In order the develop more permanent SCHEME code for TeXmacs, the simplest method is to create a personal boot file that should be named

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

We recall that `$TEXMACS_HOME_PATH` usually points to the `.TeXmacs` subdirectory of your home directory ~. If it exists, then the personal boot file is assumed to contain a SCHEME program that will be executed every time you launch TeXmacs. As a first test, you may create a personal boot file with a single line

```
(display* "Hello world!\n")
```

Every time you launch TeXmacs from the command line, this should display `Hello world!` on your terminal.

14.2.2 Adding your own keyboard shortcuts

You may use the command `kbd-map` in order to define new keyboard shortcuts. For example, if you wish to be able to start a new definition by typing `D` `e` `f` `.` (and similarly for lemmas, propositions, and theorems), then it suffices to add the following code to your personal boot file:

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

Each shortcut is of the form `(s cmd)`, where `s` is a string of key-combinations and `cmd` the SCHEME command that should be executed. The command `(make name)` starts a new tag with a given `name` at the current cursor position.

It is also possible to define shortcuts that only apply under certain circumstances. Assume for instance that we wish to define the shortcut `p` `i` for π, but only in math mode. Assume in addition that we wish to be able to quickly enter π^2 and $2\pi\mathrm{i}$ as variants. This can be achieved as follows:

```
(kbd-map
  (:mode in-math?)
  ("p i" "<mathpi>")
  ("p i var" (insert '(concat "<mathpi>" (rsup "2"))))
  ("p i var var" "2*<mathpi>*<mathi>"))
```

In this example, the command `(insert t)` is used to insert a given TeXmacs tree `t` at the current cursor position. Since the insertion of strings is very common, it is allowed to use `("p i" "<mathpi>")` as a shorthand for `("p i" (insert "<mathpi>"))`.

Special keys such as `↩`, `⌫`, `→`, etc. carry special names `return`, `backspace`, `right`, etc. Keyboard combinations using "Shift", "Control", "Alter", and "Meta" are obtained using the prefixes `S-`, `C-`, `A-`, and `M-`. In order to define `⌘^↩` as a keyboard shortcut for inserting a big vertical space, one may therefore use

```
(kbd-map
  ("C-M-return" (insert '(vspace "2fn"))))
```

Some simple keyboard shortcuts such as `→` or `^↩` are so heavily overloaded that TeXmacs treats them in a special way. For example, `→` triggers the action `(kbd-right)`, which in turn invokes the function `kbd-horizontal` with suitable arguments. Instead of redefining the `→` shortcut, it is therefore better to customize the function `kbd-horizontal` whenever appropriate. The customization of existing functions will be discussed in more detail in section 14.3.2 below.

14.2.3 Adding your own menus

$\TeX_{\text{MACS}}$ menus are generated through special SCHEME functions such as `texmacs-menu`, `insert-menu`, or `texmacs-extra-menu`. Simple menus are created using the macro `menu-bind`:

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

Simple menu entries are of the form `(label cmd)`, whereas `->` is used in order to create submenus. The menu `hello-menu` needs to be attached to one of the standard menus in order to be accessible through the interface. Top-level menus for your personal use should be added to `texmacs-extra-menu` (so that they will appear just before the Focus menu):

```
(menu-bind texmacs-extra-menu
  (=> "Hello" (link hello-menu))
  (former))
```

The syntax => corresponds to a top-level pulldown submenu, whereas `former` refers to the previous value of `texmacs-extra-menu`. We will come back to the `former` primitive in section 14.3.2 below. Here it provides us with a clean mechanism to extend the menu `texmacs-extra-menu` more than once, if needed, possibly in different SCHEME modules.

You may use `assuming` or `when` for conditional menus; in the case of `when`, the conditional menu items are greyed whenever the condition is not met. In order to add `hello-menu` to `insert-menu` under the condition that we are using the letter style, you may proceed as follows:

```
(menu-bind insert-menu
  (former)
  (assuming (style-has? "letter-style")
    --- ;; horizontal separating bar
    (link hello-menu)))
```

14.3 Extending the editor

14.3.1 Manipulation of active content

All $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ documents or document fragments can be thought of as *trees*. There are three data types that correspond to such trees:

Scheme trees. We have already encountered the type `stree` of *scheme trees* in section 14.1.2; for instance, the scheme tree `(frac (concat "a" (rsup "2")) "b+c")` represents the formula $\frac{a^2}{b+c}$. Scheme trees can in principle be manipulated using stand-alone SCHEME programs that do not even require $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ to be installed on your computer.

$\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ trees. The type `tree` of *$\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ trees* is a data type that is hard-coded in the C++ source code of $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$. It can in particular be used to represent active document fragments that are visible in $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ windows. The `tree` type is an extension to the SCHEME language that is specific to $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ and that is only available inside the editor.

Hybrid trees. Often, it is also convenient to mix the above types of trees into a super-type `tm` of so-called *hybrid trees*. More precisely, a hybrid tree is either a string, a tree, or a list of the form `(l` $\texttt{x}_1$ `...` $\texttt{x}_\texttt{n}$`)`, where `l` is a symbol and $\texttt{x}_1$, …, $\texttt{x}_\texttt{n}$ are other hybrid trees.

Here is one example of each of these types of trees:

```
Scheme] (define st '(frac (concat "a" (rsup "2")) "b+c"))
Scheme] (define tt (stree->tree st))
Scheme] (define ht `(math (concat ,tt "+" ,tt)))
```

You may already have noticed that `tree` objects are pretty-printed inside SCHEME sessions:

```
Scheme] ht

(math (concat <tree <frac|a<rsup|2>|b+c>> "+" <tree
<frac|a<rsup|2>|b+c>>))

Scheme] (tm->tree ht)
```

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

The predicates `stree?`, `tree?`, and `tm?` can be used to test whether a given SCHEME object is a tree of type `stree`, `tree`, or `tm`:

```
Scheme] (map (lambda (p?) (map p? (list st tt ht)))
             (list stree? tree? tm?))

((#t #f #f) (#f #t #f) (#t #t #t))
```

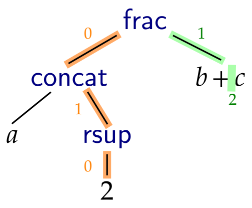
One special TeXmacs tree is the *root tree* (`root-tree`). Each open TeXmacs document is a child of the root tree and vice versa. Arbitrary subtrees of the root tree are called *active trees* and each such subtree is aware of its position inside the root tree. Using dedicated SCHEME routines from the tree API, this allows us to make changes in documents simply by assigning new values to active TeXmacs trees. In order to use these routines, one first has to import them from the appropriate module:

```
Scheme] (use-modules (utils library tree))
```

The following routine can then be implemented to clear the buffer being edited:

```
Scheme] (define (clear-document)
          (tree-set (buffer-tree) `(document "")))
```

Positions within trees are indicated through lists of numbers called *paths* and so are cursor positions. For instance, the superscript 2 in the SCHEME tree `(frac (concat "a" (rsup "2")) "b+c")` corresponds to the path `(0 1 0)`, whereas the cursor position just behind the + in the denominator corresponds to the path `(1 2)`:



The primitives `tree-set`, `tree-ref`, and `tm-ref` can be used for the modification and retrieval of subtrees along paths:

```
Scheme] (tm-ref st 0 1 0)

"2"

Scheme] (tree-set tt 0 1 0 "3")
```

3

```
Scheme] (tm->tree `(math ,ht))
```

$$\frac{a^3}{b+c}+\frac{a^3}{b+c}$$

Another useful macro for writing editing routines is `with-innermost`: it looks for the innermost tree at the current cursor position that matches a certain predicate, assigns this tree to a local variable, and then runs some code. If the predicate cannot be matched, then nothing is done. If a tag is provided instead of a predicate, then we test whether the root of the tree is labeled by that tag.

Together with what precedes, we are now in a position to define a routine that swaps the numerator and the denominator of a fraction, and attach it to the keyboard shortcut `^%`:

```
Scheme] (define (fraction-swap)
          (with-innermost t 'frac
            (tree-set t `(frac ,(tm-ref t 1)
                               ,(tm-ref t 0)))))
Scheme] (kbd-map ("C-%" (fraction-swap)))
```

14.3.2 Contextual overloading

For large software projects, it is important that different modules can be developed as independently as possible. The mechanism of *contextual overloading* is of great help here: it allows you to implement a default version of a routine in a base module, and then customize this implementation in other modules.

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

Let us illustrate how this works on a simple example. Assume that we want to define a function `hello` that inserts "Hello" by default, but "hello()" in mode math, while positioning the cursor between the brackets, after the sixth character. Using contextual overloading, this can be done as follows:

```
(tm-define (hello)
  (insert "Hello"))

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

The keyword `:mode` specifies that the second declaration only applies in math-mode, when `(in-math?)` evaluates to `#t`. The order in which routines are overloaded is important. $\text{T}_{\text{E}}\text{X}_{\text{MACS}}$ first tries the latest (re)definition. If this definition does not satisfy the requirements, then it tries the before-last (re)definition, and so on until an implementation is found that matches the requirements. In particular, if we swap the two declarations in the above example, then the general unconditional definition of `hello` will always prevail. If the two declarations are made inside different modules, then it is up to the user to ensure that the modules are loaded in the appropriate order.