Template-Mode for GNU Emacs

Mark A. Ardis

Wang Institute of Graduate Studies
Tyngsboro, Massachusetts 01879

July 31, 1987

Abstract

This manual describes *template-mode*: a set of functions for manipulating templates of text within GNU Emacs. Some of the commands insert and expand templates, while others help you create templates for later (or immediate) use.

1. Introduction to Template-Mode

Template-mode provides commands for inserting and manipulating text. It is particularly useful for inserting standard fragments, such as boilerplate for documents or function headers for programs. There are templates for major program fragments of C, Emacs-Lisp, and Pascal, as well as standard boilerplate for Scribe and TeX documents.

The chapter titled "Tutorial" explains how to invoke *template-mode*, and gives a small example. The "Commands" chapter serves as a reference manual for *template-mode* commands. Options and hooks for modifying the behavior and appearance of templates are discussed in "Customization". The "Details" chapter describes the precise syntax of templates and placeholders. Finally, the appendices contain excerpts from the template definition files.

The simplest form of templates are rectangular blocks of text. Some templates have *placeholders* within them. A placeholder may be replaced by any object of its type. For example, *text* type placeholders may be replaced by any text string that does not contain a newline. Some templates are menu lists of other templates—you must select one of the items of the list to insert. Another form of template generates a sequence of some other template—you are asked when to stop generating.

In fact, a set of templates defines a context-free grammar. Each *Sequence* template is a production. Placeholders represent nonterminals. The act of inserting and expanding a template is the same as building the derivation tree for the nonterminal on the left side of the production. *Selection* templates represent nonterminals with more than one production. *Repetition* templates represent productions of the form '`B(AB)*`', where '`A`' is any terminal string and '`B`' is any nonterminal. There are three more types of templates: *Lexical*, *String* and *Function*. Lexical templates are used to describe symbols by regular expressions. String templates are a special case of Sequence templates that contain no placeholders (and are parsed faster). Function templates provide a mechanism for invoking any Emacs-Lisp function during template expansion.

Templates are defined by a simple syntax and stored in ASCII files. The standard templates for use by *template-mode* may be found in a directory referenced by '`tpl-load-path`'. Files ending with '`.tpl`' are template source files. Compiled versions are found in files ending with '`tpl.elc`'. You may find it useful to print the '`.tpl`' files for the languages you use.

2. Tutorial – A Short Example

In the example below, the following conventions have been used:

- The first line shows a prompt, or contains '`(No prompt)`' to show the absence of prompting.
- The second, indented line shows your response. If a carriage return (`RET`) is required, it will be included in your response.

The Appendices contain the templates described in this example.

First, visit '`sample.el`', and make sure that *Emacs-Lisp-Mode* is in effect.

```
(No prompt)
            ESC x
M-x
            load-file RET
Load file:

            template RET
(No prompt)
            ESC x
M-x
            template-mode RET
```

At this point, *template-mode* will be initialized. Completion of initialization is signalled by the presence of the symbol `Template` in the mode line. If Emacs complains about not finding '`template`', then you will need to find the absolute path name for this package. (See your computer center or a knowledgeable Emacs user.)

You may want to experiment with the template-generation commands at this point, or continue the tutorial on the next page.

(No prompt)
    ^c ^t t

generate-template: Name of template?
    package **RET**

Expand? (y or n)
    y

Expand? (y or n)
    y

Replace <text:documentation> with what?
    Sample package **RET**

Expand? (y or n)
    n

Expand? (y or n)
    n

Keep optional placeholder? (y or n)
    y

Expand? (y or n)
    n

Expand? (y or n)
    y

Expand? (y or n)
    y

Replace <textenter:function-name> with what?
    reverse **RET**

Expand? (y or n)
    y

Replace <text:arguments> with what?
    list **RET**

Expand? (y or n)
    y

Type replacement and exit with ESC-^c.
    Reverse the items in LIST. **ESC-^c**

Expand? (y or n)
    y

Position on selection and exit with Return (or ESC-^c).
    x **RET**

Expand? (y or n)
    y

Replace <text:local-variables> with what?
item **RET**

Expand? (y or n)
y

Position on selection and exit with Return (or ESC-^c).
w **SPC RET**

Expand? (y or n)
y

Replace <text:list> with what?
list **RET**

Expand? (y or n)
y

Replace <text:item> with what?
item **RET**

More instances of functions? (y or n)
n

Done. ...(You may finish this function now or continue)...

(No prompt)
^c ^t ^p

(No prompt)
^c ^t ^k

(No prompt)
ESC <

Mark set
^c ^t ^e

Expand? (y or n)
y

Expand? (y or n)
y

For a shorter (i.e., fewer keystrokes) incantation of this example, try setting two options:

`tpl-keep-optional-placeholders`
: Setting this to `nil` will automatically delete optional placeholders in templates.

`tpl-ask-expansion-depth`
: Setting this to `4` will automatically begin expansion of placeholders, until the depth of recursive expansions reaches 4.

Expanding the `package` template should take fewer keystrokes after these changes.

3. Commands – How to Use Templates While Editing

This chapter serves as a reference to all of the *template-mode* commands. The first section briefly describes each command. Following sections cover commands by topic, such as selecting or expanding templates.

3.1 Summary – Short description of all the commands

Here is a short description of all of the *template-mode* commands:

`add-symbol (^c ^t a)`
: Add the symbol before point to the list of symbols created by *textenter* placeholders.

`compile-templates (unbound)`
: Store the templates in a form that allows faster loading.

`delete-placeholder (^c ^t ^k)`
: Delete the placeholder at point.

`describe-template-mode (^c ^t ^h)`
: Give a brief description of template-mode.

`expand-placeholder (^c ^t e)`
: Expands the placeholder at point with its template description.

`expand-placeholders-in-region (^c ^t ^e)`
: Expands all placeholders in the region with their template descriptions.

`expand-symbol (^c ^t TAB)`
: Expand the symbol before point using the list of symbols created by *textenter* placeholders.

`generate-any-template (^c ^t ?)`
: Builds a list of all the templates available, and waits for selection.

`generate-template (^c ^t t)`
: Prompts for the name of a template and then inserts it at point. Placeholders within the template are expanded interactively.

`load-tpl-buffer (unbound)`
: Add a new buffer of templates to the working set. (Only understands text form of templates. Uses standard buffer name completion, with default '`new.tpl`'.)

`load-tpl-file (unbound)`
: Add a new file of templates to the working set. (Only understands text form of templates, and uses standard file name completion.)

load-tpl-library (unbound)
: Add a new file of templates to the working set. (Understands both text and compiled forms of templates, and uses the *tpl-load-path* variable to find the file.)

looking-at-tpl (unbound)
: Returns *true* if the named template follows point.

next-placeholder (^c ^t ^n)
: Move to the beginning of the next placeholder.

previous-placeholder (^c ^t ^p)
: Move to the beginning of the previous placeholder.

query-replace-groups (^c ^t g)
: Replace instances of groups of lines (matched by regular expressions) with placeholders.

query-replace-lines (^c ^t l)
: Replace lines with placeholders.

query-replace-tpl (unbound)
: Replace instances of one template with corresponding instances of another.

region-to-tpl (unbound)
: Define a template, using the current region of text.

replace-line-with-placeholder (^c ^t r)
: Replace the current line with a placeholder.

replace-region-with-placeholder (^c ^t ^r)
: Replace the current region with a placeholder.

replace-tpl (unbound)
: Replace one instance of a template with a corresponding instance of another.

rewrap-template-around-point (^c ^t ^u)
: Unwrap and then wrap. (See those two commands.)

search-forward-tpl (unbound)
: Search for an instance of a template.

template-mode (unbound)
: Turns *template-mode* on and off.

unwrap-template-around-point (^c ^t u)
: Find the nearest enclosing instance of the named template and remove it, keeping the text that matches its destination placeholder.

wrap-template-around-line (^c ^t W)
: Use the current line to replace the destination placeholder of a placeholder.

wrap-template-around-region (^c ^t ^w)
: Use the current region to replace the destination placeholder of a placeholder.

wrap-template-around-word (^c ^t w)
: Use the current word to replace the destination placeholder of a placeholder.

3.2 Loading templates – initialization of template-mode

`load-tpl-buffer (unbound)`
: Add a new buffer of templates to the working set. (Only understands text form of templates. Uses standard buffer name completion, with default '`new.tpl`'.)

`load-tpl-file (unbound)`
: Add a new file of templates to the working set. (Only understands text form of templates, and uses standard file name completion.)

`load-tpl-library (unbound)`
: Add a new file of templates to the working set. (Understands both text and compiled forms of templates, and uses the *tpl-load-path* variable to find the file.)

Before a template may be used, it must be loaded. Standard templates are loaded by checking the major mode of the current buffer. (E.g., the '`ctpl.elc`' file is loaded if the major mode is "C".) You may load as many template files as you wish, in as many buffers as you wish. Each *template-mode* buffer has its own set of template files. Since many template files use similar names for templates (e.g., '`stmt`' occurs in most), it is probably a good idea to start with only the standard template files at first.

Although templates are created as text, they are converted into an internal Lisp structure when they are loaded. This conversion process may be saved by compiling the templates and storing the compiled form. The **load-tpl-library** command is capable of reading these compiled forms, which is a faster method than reading the ASCII text forms. The **load-tpl-file** and **load-tpl-buffer** commands only understand ASCII text forms. Also, **load-tpl-library** uses the value of **tpl-load-path** to find the file, whereas **load-tpl-file** and **load-tpl-buffer** use standard name completion.

3.3 Selecting templates – choosing the right template

`generate-any-template (^c ^t ?)`
: Builds a list of all the templates available, and waits for selection.

`generate-template (^c ^t t)`
: Prompts for the name of a template and then inserts it at point. Placeholders within the template are expanded interactively.

The **generate-any-template** (^c ^t ?) command builds a list of all the available templates and waits for a selection. Since the list may be quite long, it is probably best to select via the unique-prefix method: Type enough of the beginning of a template name to uniquely identify it. Point will be positioned as if an incremental search were being performed. Typing a `RET` will select an entry.

Once the selection has been made, generation proceeds by inserting a placeholder and expanding it.

The **generate-template** (^c ^t t) command requests the name of a template to insert and expand. Unambiguous prefixes may be completed with the `TAB` key. If no such template exists, you will get an error message, and an error result. Otherwise, a placeholder for the requested template will be inserted and expansion will begin.

3.4 Expanding templates – replacing placeholders by templates

`delete-placeholder (^c ^t ^k)`
Delete the placeholder at point.

`expand-placeholder (^c ^t e)`
Expands the placeholder at point with its template description.

`expand-placeholders-in-region (^c ^t ^e)`
Expands all placeholders in the region with their template descriptions.

`next-placeholder (^c ^t ^n)`
Move to the beginning of the next placeholder.

`previous-placeholder (^c ^t ^p)`
Move to the beginning of the previous placeholder.

Expansion of a placeholder consists of:

1. Finding the appropriate template.
2. Inserting the template in place of the placeholder.
3. Determining the appropriate final destination of *point* (including removing all destination placeholders).
4. Recursively expanding each placeholder within the new text.
5. Leaving *point* at the appropriate final destination.

You may choose not to expand any placeholder, in which case it is left as-is in the buffer. The command **expand-placeholder** (^c ^t e) may be used to expand the placeholder at the current location of point. The commands **next-placeholder** (^c ^t ^n), **previous-placeholder** (^c ^t ^p), and **delete-placeholder** (^c ^t ^k) are also useful when cleaning up unexpanded placeholders. **Expand-placeholders-in-region** (^c ^t ^e) will expand all of the placeholders in a region, recursively.

Placeholders may be optional or required. An optional placeholder has an extra '`#`' character at the beginning of its type name. When expanding a template containing an optional placeholder you are asked whether you wish to keep the placeholder. If you respond negatively, the placeholder is deleted. If you respond positively, the placeholder is changed into a required placeholder (i.e., the extra '`#`' is removed). For each required placeholder you are asked if you wish to expand it.

The type of a placeholder (the part before the '`:`') may be a template name, one of the *text* types (*text*, *textenter* or *textlong*), or the special *destination* type. Expansion of *text* type placeholders is discussed in a later section. If the type is a template name, that template will be inserted in place of the placeholder, and each of its placeholders will be expanded. (If the template cannot be found an error will result.) Destination placeholders are removed after inserting the containing template. They are used to select the final destination of *point*. (The location of the first destination placeholder is used if at least one such placeholder is found, otherwise the end of the template is used for the final destination.) Notice that expansion of placeholders and templates can be nested.

3.5 Menu-mode – selection of choices

If the template you request is of type *selection*, a menu will be constructed in a separate window, and you will be asked to pick one of the entries. You may move about in this window with any of the normal movement commands, but you will be prevented from modifying the buffer. The `SPC` key advances to the next line. Alphabetic characters invoke an incremental search that is anchored at the left side of the line. Thus, you may go to a particular entry by typing an unambiguous prefix of that entry.

To exit the menu type a `RET`. (Actually, **exit-recursive-edit** will work, also.) If you do not like any of the choices, type a `^g` to abort the command. You can use **advertised-undo** to clean up.

3.6 Symbol-mode – short expressions as parameters to templates

`add-symbol (^c ^t a)`
: Add the symbol before point to the list of symbols created by *textenter* placeholders.

`expand-symbol (^c ^t TAB)`
: Expand the symbol before point using the list of symbols created by *textenter* placeholders.

Expansion of a *text* placeholder is accomplished by insertion-in-place. In fact, a special mode is used for this, which restricts text modification commands to operate within a single line. The

`TAB` key is used to invoke completion. It works similarly to command completion, except that it uses an identifier table constructed by *textenter* placeholder expansions. A newline finishes the entry and exits the mode. Once the value of a *text* placeholder has been completed, every identical placeholder within the current template expansion is replaced by this value.

Expansion of a *textenter* placeholder is the same as for *text*, except that the entry is saved in the identifier table when it is completed. You may add entries to this table with the **add-symbol** command (`^c a`)—it adds the symbol before point to the table. **Expand-symbol** (`^c TAB`) expands partial symbols using this table.

3.7 Textlong – large blocks of text as parameters to templates

Expansion of a *textlong* placeholder creates a separate window for creation of the entry. You must use **exit-recursive-edit** (`ESC ^c`) to complete the entry and return to the original buffer.

3.8 Wrapping – weak parsing to manipulate enclosing templates

`rewrap-template-around-point (^c ^t ^u)`
: Unwrap and then wrap. (See those two commands.)

`unwrap-template-around-point (^c ^t u)`
: Find the nearest enclosing instance of the named template and remove it, keeping the text that matches its destination placeholder.

`wrap-template-around-line (^c ^t W)`
: Use the current line to replace the destination placeholder of a placeholder.

`wrap-template-around-region (^c ^t ^w)`
: Use the current region to replace the destination placeholder of a placeholder.

`wrap-template-around-word (^c ^t w)`
: Use the current word to replace the destination placeholder of a placeholder.

The wrap and unwrap commands provide a weak form of parsing that arises often: the recognition of enclosing templates. For example, many document processors require begin/end pairs of commands or delimiters to change the nature of processing within a region. These pairs are easily encoded as templates, with an empty middle part represented by a destination placeholder. This form of template is called a *wrapper*.

To wrap a region of text, use any of the commands: **wrap-template-around-line** (^c ^t W), **wrap-template-around-region** (^c ^t ^w), or **wrap-template-around-word** (^c ^t w). The line and word forms take a prefix argument of the number of lines or words (default = 1). To unwrap a region of text, use the **unwrap-template-around-point** (^c ^t u) command. Note that this command tries to understand indentation, but may guess incorrectly. The **rewrap-template-around-point** (^c ^t ^u) command combines unwrapping with wrapping a region. It is particularly useful when editing existing text to change the formatting style.

3.9 Searching – finding and replacing template instances

`looking-at-tpl (unbound)`
: Returns *true* if the named template follows point.

`search-forward-tpl (unbound)`
: Find the next occurrence of a template.

`replace-tpl (unbound)`
: Replace an instance of a template.

`query-replace-tpl (unbound)`
: Replace instances of templates, querying for actions on each instance.

Templates may be used in search and replacement operations, similarly to regular expressions. There are currently no backward-search operations. All of these commands take template names for arguments. So, you will have to define any non-standard templates (i.e., any templates not in the standard libraries) before using them.

Looking-at-tpl returns `t` if *point* is immediately followed by an instance of the specified template. Otherwise, it returns `nil`. **Search-forward-tpl** advances point to the end of the next instance of the named template. If no match is found, point advances to the end of the buffer. The matching algorithm assumes that templates are rectangular, and uses indentation for clues in parsing. If the text to be matched differs significantly in indentation from the template, then the matching algorithm may be fooled or confused.

Replace-tpl replaces the source template instance at point with a corresponding instance of the object template. Currently, the interpretation of "corresponding" is:

- For each placeholder in the object template, the first identically-named placeholder from the source template is used.
- Indentation of the text that matches a placeholder is adjusted rigidly to its new position.

The method of parsing templates is quite primitive, but effective. Literal characters are matched exactly, whitespace is matched loosely, and placeholders are matched by assistance of the user. (The user is asked to position *point* at the end of the matching instance.)

Query-replace-tpl behaves like **query-replace**, except that **search-forward-tpl** and **replace-tpl** are used instead of **search-forward** and **replace-string**.

3.10 Aborting – how to stop or quit

If at any time you wish to abort, type ^g. This will stop the current command and return you to normal editing. You may use *advertised-undo* (^x u) to restore the buffer to a reasonable state.

If you appear to be in a recursive-edit (denoted by '`[...]`' around part of the mode line), use **exit-recursive-edit** (ESC ^c) to exit.

3.11 Creating – how to create your own templates

`compile-templates (unbound)`
: Store the templates in a form that allows faster loading.

`query-replace-groups (^c ^t g)`
: Replace instances of groups of lines (matched by regular expressions) with placeholders.

`query-replace-lines (^c ^t l)`
: Replace lines with placeholders.

`region-to-tpl (unbound)`
: Define a template, using the current region of text.

`replace-line-with-placeholder (^c ^t r)`
: Replace the current line with a placeholder.

`replace-region-with-placeholder (^c ^t ^r)`
: Replace the current region with a placeholder.

One way to create new templates is with **replace-region-with-placeholder** (^c ^t ^r) or **replace-line-with-placeholder** (^c ^t r). These commands define a new template with the current region or line as the template body. You may edit this template by moving to the '`new.tpl`' buffer (or whatever name you supplied when requested). If you change the template after creating it, you must save the buffer and reload the file in order to use the new definition. **Region-to-tpl** may be used to begin creation of a template with the current region.

Query-replace-lines (^c ^t l) and **query-replace-groups** (^c ^t g) may be used to iteratively perform the actions of **replace-region-with-placeholder** for lines or groups of lines matching a given regular expression. These commands are particularly valuable for moving blocks of text as templates. For example, you can replace function definitions by placeholders with the same names as the functions. Sorting the placeholders and expanding results in a file of functions sorted by function names.

Since a template file is just a special ASCII file, you may create it any way you wish. There is no need to compile template files. If you do use **compile-templates**, keep in mind that only **load-tpl-library** understands this special form. Compiling templates results in faster loading, but no faster generation or expansion.

A stack of template files is kept for each buffer. So, you may define your own personal versions of templates in a file and load them after invoking *template-mode*. The newly loaded versions (your versions) will over-ride the previous definitions. There is currently no method available for "unloading" template files.

4. Customization of Templates and Their Commands

Options for customizing *template-mode* fall into four categories: command behavior, display strings, file names, and template syntax. (See the GNU Emacs Manual for help in modifying options, which are also called "variables".) Most *template-mode* options are global variables.

4.1 Behavior – options for customizing command behavior

Options in this category are used to tailor the behavior of template generation and expansion.

`tpl-ask-expansion-depth (1)`
: Depth of recursive placeholder expansions at which to start asking whether to expand. Defaults to 1, which means always ask.

`auto-load-new-templates (nil)`
: Option to automatically load new templates created with **`replace-with-placeholder`**. Otherwise, **`load-tpl-file`** must be used.

`auto-save-new-templates (nil)`
: Option to automatically save new templates created with **`replace-with-placeholder`**. Otherwise, **`save-buffer`** must be used.

`tpl-expand-placeholders (t)`
: Template-mode expansion option: if non-nil (default=t), expand placeholders after inserting them.

`tpl-fill-while-unscanning (nil)`
: Option to fill (if auto-fill is turned on) while unscanning (inserting) templates. Otherwise, templates are inserted as-is.

`tpl-form-placeholder-name-from-context (nil)`
: Option to generate placeholder names by looking for the first symbol after point. If nil, temporary names are used, instead. (This option is useful when invoking **`query-replace-groups`**.)

`tpl-get-placeholder-name-in-context (t)`
: Option to allow the user to create placeholder names in context (using *Symbol* mode). Otherwise, temporary names are used.

`tpl-include-prefix-in-groups (t)`
: Option to include the prefix string used to identify groups (when using **`query-replace-groups`**) in the group. Otherwise, the group begins *after* the identifying prefix string.

`tpl-indentation-size (2)`
: Size of indentation units in columns.

`tpl-keep-optional-placeholders ("ask")`
: Option to determine processing of optional placeholders in template-mode. If t, then always keep them. If nil, then always delete them. If neither t nor nil, then always ask.

`tpl-literal-whitespace (nil)`
: Option to preserve leading whitespace when defining new templates. Otherwise, calculate relative indentation units.

`tpl-rebuild-all-templates-template (nil)`
: Option to rebuild the list of all templates after loading new templates (including initial loading). Otherwise, the list must be built on demand.

`tpl-save-identifier-file (nil)`
: Option to keep identifier table (created with "textenter" placeholders) in a file.

`tpl-verify-end-of-group (nil)`
: Option to verify (by positioning *point*) the end of each group when using **`query-replace-groups`**.

4.2 Display – special display strings

Options in this category are used to tailor the appearance of delimiters created by *template-mode*.

`sym-end-display ("<<")`
: Display marker after string to be created in symbol mode.

`sym-start-display (">>")`
: Display marker before string to be created in symbol mode.

`tpl-display-begin (">>")`
: Delimiter marking beginning of a selected placeholder.

`tpl-display-end ("<<")`
: Delimiter marking end of a selected placeholder.

4.3 Files – names of files

Options in this category consist of file and directory names used by *template-mode*.

`auto-template-alist (nil)`
: Global list of major modes and their associated template files. It is initialized by **`tpl-initialize-modes`**.

`tpl-load-path (nil ...)`
: A list of directories to search when looking for templates. Nil means look in the current directory.

`tpl-new-template-buffer (‘‘new.tpl’’)`
: Buffer containing new templates.

4.4 Syntax – patterns and strings in template definitions

Options in this category consist of regular expressions and strings used to define templates. There are several options that may be changed, but most should be left alone. Since regular expression search is used quite often, one should not use characters that are special to regular expressions in delimiters. For example, the use of ‘*’ in any of the delimiters is sure to cause problems.

`tpl-begin-placeholder (‘‘<’’)`
: Regular expression for beginning of placeholder.

`tpl-begin-template-body (‘‘^:begin’’)`
: Regular expression for beginning of template body.

`tpl-begin-template-definition (‘‘^Template’’)`
: Regular expression for beginning of template definition.

`tpl-destination-symbol (‘‘POINT’’)`
: Special placeholder name used as destination of point after expansion.

`tpl-end-placeholder (‘‘>’’)`
: Regular expression for end of placeholder.

`tpl-end-template-body (‘‘^:end’’)`
: Regular expression for end of template body.

`tpl-function-type (‘‘Function’’)`
: Name of function-type template type.

`tpl-lexical-type (‘‘Lexical’’)`
: Name of lexical-type template type.

`tpl-next-placeholder-number (1)`
: Counter used to generate temporary placeholder names.

`tpl-pattern-optional (‘‘#’’)`
: Regular expression for optional placeholder delimiter.

`tpl-pattern-other (‘‘.’’)`
: Regular expression for all other tokens.

`tpl-pattern-placeholder (too complicated to print)`
: Regular expression for placeholder—built from component parts above.

`tpl-pattern-punctuation (‘‘\\s.+’’)`
: Regular expression for at least one punctuation character.

`tpl-pattern-string (.*)`
: Regular expression for each line of string-type templates.

`tpl-pattern-symbol (‘‘\\(\\sw\\|\\s_\\)+’’)`
: Regular expression for at least one symbol character.

`tpl-pattern-whitespace (too complicated to print)`
: Regular expression for at least one whitespace character.

`tpl-pattern-word (‘‘\\sw+’’)`
: Regular expression for at least one word character.

`tpl-repetition-type (‘‘Repetition’’)`
: Name of repetition-type template type.

`tpl-selection-type (‘‘Selection’’)`
: Name of selection-type template type.

`tpl-sep-placeholder (‘‘:’’)`
: Regular expression for separator within placeholder.

`tpl-sequence-type (‘‘Sequence’’)`
: Name of sequence-type template type.

`tpl-string-type (‘‘String’’)`
: Name of string-type template type.

`tpl-temporary-placeholder-name (‘‘TEMP’’)`
: Root of temporary placeholder names.

4.5 Hooks – user-defined functions to change behavior

`sym-check-validity-hook`
: This hook may be used to check the validity of expansions of *text* and *textenter* placeholders. It is temporarily replaced by an appropriate function for *lexical* placeholders.

`template-mode-hook`
: This hook is called when *template-mode* is turned on.

`template-mode-load-hook`
: This hook is called when *template-mode* is loaded. (Loading includes initialization of the global list of template files to use for different modes. This hook is useful for altering that list.)

`template-scan-hook`
: This hook is called before attempting to scan for an instance of a template. Reformatting with this hook may make it easier to find instances of templates.

`template-unscan-hook`
: This hook is called after unscanning (inserting) a template. Reformatting with this hook may be useful.

5. Details – Template and Placeholder Syntax

This chapter covers all the details of template syntax for those who wish to create their own template files. Note that the concrete syntax for template definitions may be changed through several user options (e.g., **tpl-begin-template-body**). These options are usually local variables, so that different template files may have different syntax. However, it is best not to modify the syntax of placeholders (even though there are options for doing this), since placeholders usually span template files.

A template file consists of a sequence of individual template definitions. Each definition has the form:

```
Template <template-name> <template-type>
:begin
<template-body>
:end
```

Note that the newline at the very end of the template definition is *not* part of the template body. The contents and interpretation of the template body differ depending on template type. In general, literal strings in the body stand for themselves. There are no escape characters for including special characters.

Placeholders have a special syntax:

"<" <placeholder-type> [: <placeholder-name>] ">"

(where the square brackets indicate optional parts). Placeholder names are usually used to prompt the user for suggestive replacements or context, but are especially significant for **text** placeholders. Identical **text** placeholders within a template are replaced with the same value. Two placeholders are identical only if all of their fields are identical, including their names (if present).

As mentioned earlier, there are four special forms of placeholders:

text May be replaced by any text not containing a newline.

textenter
Similar to **text**, but saves the result in the identifier completion table.

textlong May be replaced by any text, including newlines.

destination (point)
Represents the intended final destination of *point*. These placeholders are always removed from the template body after expanding.

All other placeholders must define templates. There are six forms of templates:

Function The result of executing the body of the template (written in GNU Emacs Lisp) is inserted in place of the placeholder.

Lexical Similar to **text**, but checks the result against a regular expression. The body of the template must be a valid GNU Emacs Lisp regular expression.

Repetition
These templates must be of the form: "<separator string> <placeholder>". They result in a series of expansions of "<placeholder>", separated by "<separator string>"s. Note that a newline may be used for the separator.

Selection
The template body must consist of a list of items of the following form:

"<selection> [: [<placeholder>]] [; comment]"

(where square brackets indicate optional parts). If there is no ":", then the "<selection>" is considered to be a string replacement for the placeholder. If there is a ":" then the item is considered to be a reference to another placeholder. In that case, if there is a "<placeholder>" it is used for the replacement, otherwise the "<selection>" is used.

Sequence This type of template may have literal text and/or placeholders in its body.

String This type is similar to **sequence**, but may not contain placeholders. It is parsed more simply (and faster).

The best method for creating new template files is to start with an existing template file. There are several examples of **function** type templates in the generic template file. Examples of other types are present in the Emacs-Lisp template file.

Appendix A. Example Templates for Emacs-Lisp

```
Template arg-type Selection
:begin
a ; Function name
b ; Buffer name (must exist)
B ; Buffer name (possibly nonexistent)
c ; Single character
C ; Command name
d ; Point as a number (no prompt)
D ; Directory name
f ; File name (must exist)
F ; File name (possibly nonexistent)
k ; Keystroke sequence (string)
m ; Mark as a number (no prompt)
n ; Number (reads a string and converts)
p ; Prefix arg converted to number
P ; Prefix arg in raw form
r ; Region (no prompt)
s ; String
S ; Symbol
v ; Variable name (must be user-variable-p)
x ; Lisp expression unevaluated
X ; Lisp expression evaluated
:end

Template function Sequence
:begin

(defun <textenter:function-name> (<text:arguments>)
  "<textlong:documentation>"
  (interactive "<arg-type><textenter:function-name>: <text:arguments>? ")
; Local Variables
  (let (<text:local-variables>)
; Body
    <stmt:body>
  ) ; let
) ; defun <textenter:function-name>

:end

Template functions Repetition
:begin

<function>
:end
```

```
Template package Sequence
:begin
;;; <buffer-name> -- <text:documentation>
;;; <user-full-name>, <today>

<#variables>
<functions>
:end

Template stmt Selection
:begin
progn:
cond:
debug:
ife:
if:
while:
whilelist:
simple:
:end

Template whilelist Sequence
:begin
(while <text:list>
  (setq <text:item> (car <text:list>))
  (setq <text:list> (cdr <text:list>))
  <POINT>
) ; while <text:list>
:end
```

Appendix B. Example Generic Templates

```
Template buffer-name Function
:begin
(buffer-name)
:end

Template day Function
:begin
(let ((str (current-time-string)))
  (string-match "[0-9]+" str)
  (substring str (match-beginning 0) (match-end 0))
)
:end

Template month Function
:begin
(let ((str (current-time-string))
      prefix)
  (setq prefix (substring str 4 7))
  (cond
    ((equal prefix "Jan")
      "January")
    ((equal prefix "Feb")
      "February")
    ((equal prefix "Mar")
      "March")
    ((equal prefix "Apr")
      "April")
    ((equal prefix "May")
      "May")
    ((equal prefix "Jun")
      "June")
    ((equal prefix "Jul")
      "July")
    ((equal prefix "Aug")
      "August")
    ((equal prefix "Sep")
      "September")
    ((equal prefix "Oct")
      "October")
    ((equal prefix "Nov")
      "November")
    ((equal prefix "Dec")
      "December")
  )
)
:end
```

```
Template today Function
:begin
(concat "<month>" " " "<day>" ", 19" "<year>")
:end

Template user-full-name Function
:begin
(user-full-name)
:end

Template year Function
:begin
(substring (current-time-string) -2)
:end
```

Table of Contents