MMM Mode Manual

Multiple Major Modes for Emacs
Edition 0.4.8
9 March 2003

Michael Abraham Shulman

Copyright © 2000 Michael Abraham Shulman.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled "Copying" and "GNU General Public License" are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

1 Overview of MMM Mode

MMM Mode is a minor mode for Emacs which allows Multiple Major Modes to coexist in a single buffer. The name is an abbreviation of 'Multiple Major Modes'[1]. A major mode is a customization of Emacs for editing a certain type of text, such as code for a specific programming language. See Section "Major Modes" in *The Emacs Manual*, for details.

MMM Mode is a general extension to Emacs which is useful whenever one file contains text in two or more programming languages, or that should be in two or more different modes. For example:

- CGI scripts written in any language, from Perl to PL/SQL, may want to output verbatim HTML, and the writer of such scripts may want to use Emacs' html-mode or sgml-mode to edit this HTML code, while remaining in the appropriate programming language mode for the rest of the file. See Section 4.3 [Here-documents], page 16, for example.
- There are now many "content delivery systems" which turn the CGI script idea around and simply add extra commands to an HTML file, often in some programming language, which are interpreted on the server. See Section 4.1 [Mason], page 15, See Section 4.6 [Embperl], page 17, See Section 4.7 [ePerl], page 17, See Section 4.8 [JSP], page 18.
- HTML itself can also contain embedded languages such as Javascript and CSS styles, for which Emacs has different major modes. See Section 4.4 [Javascript], page 17, and See Section 4.5 [Embedded CSS], page 17, for example.
- The idea of "literate programming" requires the same file to contain documentation (written as text, html, latex, etc.) and code (in an appropriate programming language). See Section 4.10 [Noweb], page 18, for example.
- Emacs allows files of any type to contain 'local variables', which can include Emacs Lisp code to be evaluated. See Section "File Variables" in *The Emacs Manual*. It may be easier to edit this code in Emacs Lisp mode than in whatever mode is used for the rest of the file. See Section 4.2 [File Variables], page 16.
- There are many more possible uses for MMM Mode. RPM spec files can contain shell scripts (see Section 4.9 [RPM], page 18). Email or newsgroup messages may contain sample code. And so on. We encourage you to experiment.

1.1 Basic Concepts

The way MMM Mode works is as follows. Each buffer has a *dominant* or *default* major mode, which is chosen as major modes normally are: the user can set it interactively, or it can be chosen automatically with 'auto-mode-alist' (see Section "Choosing Modes" in *The Emacs Manual*). Within the file, MMM Mode creates *submode regions* within which other major modes are in effect. While the point is in a submode region, the following changes occur:

1. The local keymap is that of the submode. This means the key bindings for the submode are available, while those of the dominant mode are not.

[1] The name is derived from `mmm.el` for XEmacs by Gongquan Chen <chen@posc.org>, from which MMM Mode was adapted.

2. The mode line (see Section "Mode Line" in *The Emacs Manual*) changes to show which submode region is active. This can be configured; see Section 3.3 [Mode Line], page 11.
3. The major mode menu, both on the menu bar and the mouse popup, are that of the submode.
4. Some local variables of the submode shadow those of the default mode (see Section 3.5 [Local Variables], page 12). For the user, this serves to help make Emacs behave as if the submode were the major mode.
5. The syntax table and indentation are those of the submode.
6. Font-lock (see Section "Font Lock" in *The Emacs Manual*) fontifies correctly for the submode.
7. The submode regions are highlighted by a background color; see Section 3.1 [Region Coloring], page 10.

The submode regions are represented internally by Emacs Lisp objects known as *overlays*. Some of the above are implemented by overlay properties, and others are updated by an MMM Mode function in 'post-command-hook'. You don't need to know this to use MMM Mode, but it may make any error messages you come across more understandable. See Section "Overlays" in *The GNU Emacs Lisp Reference Manual*, for more information on overlays.

Because overlays are not saved with a file, every time a file is opened, they must be created. Creating submode regions is occasionally referred to as *mmm-ification*. (I've never had occasion to pronounce this, but if I did I would probably say 'mummification'. Like what they did in ancient Egypt.) You can mmm-ify a buffer interactively, but most often MMM Mode will find and create submode regions automatically based on a buffer's file extension, dominant mode, or local variables.

1.2 Installing MMM Mode

MMM Mode has a standard installation process. See the file INSTALL for generic information on this process. To summarize, unpack the archive, `cd` to the created MMM Mode directory, type '`./configure`', then '`make`', then '`make install`'. If all goes correctly, this will compile the MMM Mode elisp files, install them in your local site-lisp directory, and install the MMM Mode info file `mmm.info` in your local info directory.

Now you need to configure your Emacs initialization file (usually `~/.emacs`) to use MMM Mode. First, Emacs has to know where to find MMM Mode. In other words, the MMM Mode directory has to be in `load-path`. This can be done in the parent directory's `subdirs.el` file, or in the init file with a line such as:

```
(add-to-list 'load-path "/path/to/site-lisp/mmm/")
```

Once `load-path` is configured, MMM Mode must be loaded. You can load all of MMM Mode with the line

```
(require 'mmm-mode)
```

but if you use MMM Mode only rarely, it may not be desirable to load all of it at the beginning of every editing session. You can load just enough of MMM Mode so it will turn itself on when necessary and load the rest of itself, by using instead the line

```
(require 'mmm-auto)
```

in your initialization file.

One more thing you may want to do right now is to set the variable `mmm-global-mode`. If this variable is `nil` (the default), MMM Mode will never turn itself on. If it is `t`, MMM Mode will turn itself on in every buffer. Probably the most useful value for it, however, is the symbol `maybe` (actually, anything that is not `nil` and not `t`), which causes MMM Mode to turn itself on in precisely those buffers where it would be useful. You can do this with a line such as:

```
(setq mmm-global-mode 'maybe)
```

in your initialization file. See Section 2.7 [Global Mode], page 8, for more detailed information.

1.3 Getting Started Quickly

Perhaps the simplest way to create submode regions is to do it interactively by specifying a region. First you must turn MMM Mode on—say, with *M-x mmm-mode*—then place point and mark around the area you want to make into a submode region, type *C-c % C-r*, and enter the desired major mode. See Section 2.6 [Interactive], page 8, for more details.

A better way to add submode regions is by using submode classes, which store a lot of useful information for MMM Mode about how to add and manipulate the regions created. See Section 2.2 [Submode Classes], page 5, for more details. There are several sample submode classes that come with MMM Mode, which are documented later in this manual. Look through these and determine if one of them fits your needs. If so, I suggest reading the comments on that mode. Then come back here to find out to use it.

To apply a submode class to a buffer interactively, turn MMM Mode on as above, then type *C-c % C-c* and enter the name of the class. Submode regions should be added automatically, if there are any regions in the buffer appropriate to the submode class.

If you want a given file to always use a given submode class, you can express this in a file variable: add a line containing the string '`-*- mmm-classes:` *class* `-*-`' at the top of the file. See Section "File Variables" in *The Emacs Manual*, for more information and other methods. Now whenever MMM Mode is turned on in that file, it will be mmm-ified according to *class*. If `mmm-global-mode` is non-nil, then MMM Mode will turn itself on whenever a file with a `mmm-classes` local variable is opened. See Section 2.7 [Global Mode], page 8, for more information.

If you want a submode class to apply to *all* files in a certain major mode or with a certain extension, add a line such as this to your initialization file:

```
(mmm-add-mode-ext-class mode extension class)
```

After this call, any file opened whose name matches the regular expression *extension and* whose default mode is *mode* will be automatically mmm-ified according to *class* (assuming `mmm-global-mode` is non-nil). If one of *extension* or *mode* is `nil`, a file need only satisfy the other one to be mmm-ified.

You can now read the rest of this manual to learn more about how MMM Mode works and how to configure it to your preferences. If none of the supplied submode classes fit your needs, then you can try to write your own. See Chapter 5 [Writing Classes], page 20, for more information.

2 MMM Mode Basics

This chapter explains the most important parts of how to use MMM Mode.

2.1 The MMM Minor Mode

An Emacs minor mode is an optional feature which can be turned on or off in a given buffer, independently of the major mode. See Section "Minor Modes" in *The Emacs Manual*. MMM Mode is implemented as a minor mode which manages the submode regions. This minor mode must be turned on in a buffer for submode regions to be effective. When activated, the MMM Minor mode is denoted by '`MMM`' in the mode line (see Section 3.3 [Mode Line], page 11).

2.1.1 Enabling MMM Mode

If `mmm-global-mode` is non-`nil` (see Section 2.7 [Global Mode], page 8), then the MMM minor mode will be turned on automatically whenever a file with associated submode classes is opened (see Section 2.3 [Selecting Classes], page 5). It is also turned on by interactive mmm-ification (see Section 2.6 [Interactive], page 8), although the interactive commands do not have key bindings when it is not on and must be invoked via *`M-x`*. You can also turn it on (or off) manually with *`M-x mmm-mode`*, in which case it applies all submode classes associated with the buffer. Turning MMM Mode off automatically removes all submode regions from the buffer.

`mmm-mode` *arg* [Command]
: Toggle the state of MMM Mode in the current buffer. If *arg* is supplied, turn MMM Mode on if and only if *arg* is positive.

`mmm-mode-on` [Function]
: Turn MMM Mode on unconditionally in the current buffer.

`mmm-mode-off` [Function]
: Turn MMM Mode off unconditionally in the current buffer.

`mmm-mode` [Variable]
: This variable represents whether MMM Mode is on in the current buffer. Do not set this variable directly; use one of the above functions.

2.1.2 Key Bindings in MMM Mode

When MMM Mode is on, it defines a number of key bindings. By default, these are bound after the prefix sequence *`C-c %`*. Minor mode keymaps are supposed to use *`C-c punctuation`* sequences, and I find this one to be a good mnemonic because '`%`' is used by Mason to denote special tags. This prefix key can be customized; Section 3.4 [Key Bindings], page 12.

There are two types of key bindings in MMM Mode: *commands* and *insertions*. Command bindings run MMM Mode interactive functions to do things like re-parse the buffer or end the current submode region, and are defined statically as normal Emacs key-bindings. Insertion bindings insert submode region skeletons with delimiters into the buffer, and are defined dynamically, according to which submode classes (see Section 2.2 [Submode Classes], page 5) are in effect, via a keymap default binding.

To distinguish between the two, MMM Mode uses distinct modifier keys for each. By default, command bindings use the control key (e.g. `C-c % C-b` re-parses the buffer), and insertion bindings do not (e.g. `C-c % p`, when the Mason class is in effect, inserts a '`<%perl>...</%perl>`' region). This makes the command bindings different from in previous versions, however, so the variable `mmm-use-old-bindings` is provided. If this variable is set to 't' before MMM Mode is loaded, the bindings will be reversed: insertion bindings will use the control key and command bindings will not.

Normally, Emacs gives help on a prefix command if you type `C-h` after that command (e.g. `C-x C-h` displays all key bindings starting with `C-x`). Because of how insertion bindings are implemented dynamically with a default binding, they do not show up when you hit `C-c % C-h`. For this reason, MMM Mode defines the command `C-c % h` which displays a list of all currently valid insertion key sequences. If you use the defaults for command and insertion bindings, the `C-h` and `h` should be mnemonic.

In the rest of this manual, I will assume you are using the defaults for the mode prefix (`C-c %`) and the command and insertion modifiers. You can customize them, however; Section 3.4 [Key Bindings], page 12.

2.2 Understanding Submode Classes

A submode class represents a "type" of submode region. It specifies how to find the regions, what their delimiters look like, what submode they should be, how to insert them, and how they behave in other ways. It is represented by a symbol, such as `mason` or `eval-elisp`.

For example, in the Mason set of classes, there is one class representing all '`<%...%>`' inline Perl regions, and one representing regions such as '`<%perl>...</%perl>`', '`<%init>...</%init>`', and so on. These are different to Mason, but to Emacs they are all just Perl sections, so they are covered by the same submode class.

But it would be tedious if whenever we wanted to use the Mason classes, we had to specify both of these. (Actually, this is a simplification: there are some half a dozen Mason submode classes.) So submode classes can also "group" others together, and we can refer to the `mason` class and mean all of them.

The way a submode class is used is to *apply* it to a buffer. This scans the buffer for regions which should be submode regions according to that class, and also remembers the class for later, so that new submode regions can be inserted and scanned for later.

2.3 How MMM Mode selects submode classes

Submode classes that apply to a buffer come from three sources: mode/extension-associated classes, file-local classes, and interactive MMM-ification (see Section 2.6 [Interactive], page 8). Whenever MMM Mode is turned on in a buffer (see Section 2.1 [MMM Minor Mode], page 4, and Section 2.7 [Global Mode], page 8), it inspects the value of two variables to determine which classes to automatically apply to the buffer. This covers the first two sources; the latter is covered in a later chapter.

2.3.1 File-Local Submode Classes

`mmm-classes` [Variable]
: This variable is always buffer-local when set. Its value should be either a single symbol or a list of symbols. Each symbol represents a submode class that is applied to the buffer.

`mmm-classes` is usually set in a file local variables list. See Section "File Variables" in *The Emacs Manual*. The easiest way to do this is for the first line of the file to contain the string '`-*- mmm-classes: classes -*-`', where *classes* is the desired value of `mmm-classes` for the file in question. It can also be done with a local variables list at the end of the file.

2.3.2 Submode Classes Associated with Modes and Extensions

`mmm-mode-ext-classes-alist` [User Option]
: This global variable associates certain submode classes with major modes and/or file extensions. Its value is a list of elements of the form `(mode ext class)`. Any buffer whose major mode is *mode* (a symbol) *and* whose file name matches *ext* (a regular expression) will automatically have the submode class *class* applied to it.

  If *mode* is `nil`, then only *ext* is considered to determine if a buffer fits the criteria, and vice versa. Thus if both *mode* and *ext* are nil, then *class* is applied to *all* buffers in which MMM Mode is on. Note that *ext* can be any regular expression, although its name indicates that it most often refers to the file extension.

  If *class* is the symbol `t`, then no submode class is actually applied for this association. However, if `mmm-global-mode` is non-`nil` and non-`t`, MMM Mode will be turned on in matching buffers even if there are no actual submode classes being applied. See Section 2.7 [Global Mode], page 8.

`mmm-add-mode-ext-class` *mode ext class* [Function]
: This function adds an element to `mmm-mode-ext-classes-alist`, associating the submode class *class* with the major mode *mode* and extension *ext*.

  Older versions of MMM Mode required this function to be used to control the value of `mmm-mode-ext-classes-alist`, rather than setting it directly. In this version it is provided purely for convenience and backward compatibility.

2.3.3 Globally Applied Classes and the Universal Class

In addition to file-local and mode-ext-associated submode classes, MMM Mode also allows you to specify that certain submode classes apply to *all* buffers in which MMM Mode is enabled.

`mmm-global-classes` [User Option]
: This variable's value should be a list of submode classes that apply to all buffers with MMM Mode on. It can be overriden in a file local variables list, such as to disable global class for a specific file. Its default value is `(universal)`.

The default global class is the "universal class", which is defined in the file `mmm-univ.el` (loaded automatically), and allows the author of text to specify that a certain section of it be in a specific major mode. Thus, for example, when writing an email message that

includes sample code, the author can allow readers of the message (who use emacs and MMM) to view the code in the appropriate major mode. The syntax used is '`{%mode%} ... {%/mode%}`', where *mode* should be the name of the major mode, with or without the customary '`-mode`' suffix: for example, both '`cperl`' and '`cperl-mode`' are acceptable.

The universal class also defines an insertion key, '`/`', which prompts for the submode to use. See Section 2.4 [Insertion], page 7. The universal class is most useful when `mmm-global-mode` is set to `t`; Section 2.7 [Global Mode], page 8.

2.4 Inserting new submode regions

So much for noticing submode regions already present when you open a file. When editing a file with MMM Mode on, you will often want to add a new submode region. MMM Mode provides several facilities to help you. The simplest is to just hit a few keys and have the region and its delimiters inserted for you.

Each submode class can define an association of keystrokes with "skeletons" to insert a submode region. If there are several submode classes enabled in a buffer, it is conceivable that the keys they use for insertion might conflict, but unlikely as most buffers will not use more than one or two submode classes groups.

As an example of how insertion works, consider the Mason classes. In a buffer with MMM Mode enabled and Mason associated, the key sequence `C-c % p` inserts the following perl section (the semicolon is to prevent CPerl Mode from getting confused—see Section 4.1 [Mason], page 15):

```
<%perl>-<-;
-!-
->-</%perl>
```

In this schematic representation, the string '`-!-`' represents the position of point (the cursor), '`-<-`' represents the beginning of the submode region, and '`->-`' its end.

All insertion keys come after the MMM Mode prefix keys (by default `C-c %`; see Section 3.4 [Key Bindings], page 12) and are by default single characters such as `p`, `%`, and `i`. To avoid confusion, all the MMM Mode commands are bound by default to control characters (after the same prefix keys), such as `C-b`, `C-%` and `C-r`. This is a change from earlier versions of MMM Mode, and can be customized; see Section 3.4 [Key Bindings], page 12.

To find out what insertion keys are available, consult the documentation for the submode class you are using. If it is one of the classes supplied with MMM Mode, you can find it in this Info file.

Because insertion keys are implemented with a "default binding" for flexibility, they do not show up in the output of `C-h m` and cannot be found with `C-h k`. For this reason, MMM Mode supplies the command `C-c % h` (`mmm-insertion-help` to view the available insertion keys.

2.5 Re-Parsing Submode Regions

Describe `mmm-parse-buffer`, `mmm-parse-region`, `mmm-parse-block`, and `mmm-clear-current-region`.

2.6 Interactive MMM-ification Functions

There are several commands you can use to create submode regions interactively, rather than by applying a submode class to a buffer. These commands (in particular, `mmm-ify-region`), can be useful when editing a file or email message containing a snippet of code in some other language. Also see Section 2.3.3 [Global Classes], page 6, for an alternate approach to the same problem.

C-c % C-r Creates a submode region between point and mark. Prompts for the submode to use, which must be a valid Emacs major mode name, such as `emacs-lisp-mode` or `cperl-mode`. Adds markers to the interactive history. (`mmm-ify-region`)

C-c % C-c Applies an already-defined submode class to the buffer, which it prompts for. Adds this class to the interactive history. (`mmm-ify-by-class`)

C-c % C-x Scans the buffer for submode regions (prompts for the submode) using front and back regular expressions that it also prompts for. Briefly, it starts at the beginning of the buffer and searches for the front regexp. If it finds a match, it searches for the back regexp. If it finds a match for that as well, it makes a submode region between the two matches and continues searching until no more matches are found. Adds the regexps to the interactive history. (`mmm-ify-by-regexp`)

These commands are also useful when designing a new submode class (see Section 2.2 [Submode Classes], page 5). Working with the regexps interactively can make it easier to debug and tune the class before starting to use it on automatic. All these commands also add to value of the following variable.

`mmm-interactive-history` [Variable]
Stores a history of all interactive mmm-ification that has been performed in the current buffer. This way, for example, the re-parsing functions (see Section 2.5 [Re-parsing], page 7) will respect interactively added regions, and the insertion keys for classes that were added interactively are available.

If for any reason you want to "wipe the slate clean", this command should help you. By default, it has no key binding, so you must invoke it with *M-x mmm-clear-history RET*.

`mmm-clear-history` [Command]
Clears all history of interactive mmm-ification in the current buffer. This command does not affect existing submode regions; to remove them, you may want to re-parse the buffer with *C-c % C-b* (`mmm-parse-buffer`).

2.7 MMM Global Mode

When a file has associated submode classes (see Section 2.3 [Selecting Classes], page 5), you may want MMM Mode to turn itself on and parse that file for submode regions automatically whenever it is opened in an Emacs buffer. The value of the following variable controls when MMM Mode turns itself on automatically.

`mmm-global-mode` [User Option]
Do not be misled by the fact that this variable's name ends in '`-mode`': it is not a simple on/off switch. There are three possible (meanings of) values for it: `t`, `nil`, and anything else.

When this variable is `nil`, MMM Mode is never enabled automatically. If it is enabled manually, such as by typing *M-x mmm-mode*, any submode classes associated with the buffer will still be used, however.

When this variable is `t`, MMM Mode is enabled automatically in *all* buffers, including those not visiting files, except those whose major mode is an element of `mmm-never-modes`. The default value of this variable contains modes such as `help-mode` and `dired-mode` in which most users would never want MMM Mode, and in which MMM might cause problems.

When this variable is neither `nil` nor `t`, MMM Mode is enabled automatically in all buffers that would have associated submode classes; i.e. only if there would be something for it to do. The value of `mmm-never-modes` is still respected, however. Note that this can include buffers not visiting files, if that buffer's major mode is present in `mmm-mode-ext-classes-alist` with a `nil` value for *ext* (see Section 2.3.2 [Mode-Ext Classes], page 6). Submode class values of `t` in `mmm-mode-ext-classes-alist` cause MMM Mode to be enabled in matching buffers, but supply no submode classes to be applied.

2.7.1 The Major Mode Hook

This section is intended for users who understand Emacs Lisp and want to know how MMM Global Mode is implemented, and perhaps use the same technique. In fact, MMM Mode exports a hook variable that you can use easily, without understanding any of the details—see below.

In order to enable itself in *all* buffers, however, MMM Mode has to hook itself into all major modes. Global Font Lock Mode from the standard Emacs distribution (see Section "Font Lock" in *The Emacs Manual*) has a similar problem, and solves it by adding a function to `change-major-mode-hook`, which is run by `kill-all-local-variables`, which is run in turn by all major mode functions at the *beginning*. This function stores a list of which buffers need fontification. It then adds a different function to `post-command-hook`, which checks if the current buffer needs fontification, and if so performs it. MMM Global Mode uses the same technique.

In the interests of generality, and for your use, the function that MMM Mode runs in `post-command-hook` (`mmm-run-major-mode-hook`) is not specific to MMM Mode, but rather runs the hook variable `mmm-major-mode-hook`, which by default contains a function (`mmm-mode-on-maybe`) which possibly turns MMM Mode on, depending on the value of `mmm-global-mode`. Thus, to run another function in all major modes, all you need to do is add it to this hook. For example, the following line in an initialization file will turn on Auto Fill Mode (see Section "Auto Fill" in *The Emacs Manual*) in all buffers:

```
(add-hook 'mmm-major-mode-hook 'turn-on-auto-fill)
```

3 Customizing MMM Mode

This chapter explains how to customize the appearance and functioning of MMM Mode however you want.

3.1 Customizing Region Coloring

By default, MMM Mode highlights all submode regions with a background color. There are three levels of this decoration, controlled by the following variable:

`mmm-submode-decoration-level` [User Option]
: This variable controls the level of coloring of submode regions. It should be one of the integers 0, 1, or 2, representing (respectively) none, low, and high coloring.

No coloring means exactly that. Submode regions have the same background as the rest of the text. This produces the minimal interference with font-lock coloration. In particular, if you want to use background colors for font-lock, this may be a good idea, because the submode highlight, if present, overrides any font-lock background coloring.

Low coloring uses the same background color for all submode regions. This color is specified with the face `mmm-default-submode-face` (see Section "Faces" in *The Emacs Manual*) which can be customized, either through the Emacs "customize" interface or using direct Lisp commands such as `set-face-background`. Of course, other aspects of the face can also be set, such as the foreground color, bold, underline, etc. These are more likely to conflict with font-lock, however, so only a background color is recommended.

High coloring uses multiple background colors, depending on the function of the submode region. The recognized functions and their meanings are as follows:

'`init`' Code that is executed at the beginning of (something), as initialization of some sort.

'`cleanup`' Code that is executed at the end of (something), as some sort of clean up facility.

'`declaration`'
: Code that provides declarations of some sort, perhaps global or local arguments, variables, or methods.

'`comment`' Text that is not executed as code, but instead serves to document the code around it. Submode regions of this function often use a mode such as Text Mode rather than a programming language mode.

'`output`' An expression that is evaluated and its value interpolated into the output produced.

'`code`' Executed code not falling under any other category.

'`special`' Submode regions not falling under any other category, such as component calls.

The different background colors are provided by the faces `mmm-`*`function`*`-submode-face`, which can be customized in the same way as `mmm-default-submode-face`.

3.2 Preferred Major Modes

Certain of the supplied submode classes know only the language that certain sections are written in, but not what major mode you prefer to use to edit such code. For example, many people prefer CPerl mode over Perl mode; you may have a special mode for Javascript or just use C++ mode. This variable allows you to tell submodes such as Mason (see Section 4.1 [Mason], page 15) and Embedded Javascript (see Section 4.4 [Javascript], page 17) what major mode to use for the submodes:

`mmm-major-mode-preferences` [User Option]

The elements of this list are cons cells of the form (*language* . *mode*). *language* should be a symbol such as `perl`, `html-js`, or `java`, while *mode* should be the name of a major mode such as `perl-mode`, `cperl-mode`, `javascript-mode`, or `c++-mode`.

You probably won't have to set this variable at all; MMM tries to make intelligent guesses about what modes you prefer. For example, if a function called `javascript-mode` exists, it is chosen, otherwise `c++-mode` is used. Similarly for `jde-mode` and `java-mode`.

If you do need to change the defaults, you may find the following function convenient.

`mmm-set-major-mode-preferences` *language mode* **&optional** *default* [Function]

Set the preferred major mode for LANGUAGE to MODE. If there is already a mode specified for LANGUAGE, and DEFAULT is nil or unsupplied, then it is changed. If DEFAULT is non-nil, then any existing mode is unchanged. This is used by packages to ensure that some mode is present, but not override any user-specified mode. If you are not writing a submode class, you should ignore the third argument.

Thus, for example, to use `my-java-mode` for Java code, you would use the following line:

```
(mmm-set-major-mode-preferences 'java 'my-java-mode)
```

3.3 Customizing the Mode Line Display

By default, when in a submode region, MMM Mode changes the section of the mode line (see Section "Mode Line" in *The Emacs Manual*) that normally displays the major mode name—for example, '`HTML`'—to instead show both the dominant major mode and the currently active submode—for example, '`HTML[CPerl]`'. You can change this format, however.

`mmm-submode-mode-line-format` [User Option]

The value of this variable should be a string containing one or both of the escape sequences '`~M`' and '`~m`'. The string displayed in the major mode section of the mode line when in a submode is obtained by replacing all occurrences of '`~M`' with the dominant major mode name and '`~m`' with the currently active submode name. For example, to display only the currently active submode, set this variable to '`~m`'. The default value is '`~M[~m]`'.

The MMM minor mode also normally displays the string '`MMM`' in the minor mode section of the mode line to indicate when it is active. You can customize or disable this as well.

`mmm-mode-string` [User Option]

This string is displayed in the minor mode section of the mode line when the MMM minor mode is active. If nonempty, it should begin with a space to separate the MMM indicator from that of other minor modes. To eliminate the indicator entirely, set this variable to the empty string.

3.4 Customizing the MMM Mode Key Bindings

The default MMM Mode key bindings are explained in Section 2.1.2 [MMM Mode Keys], page 4, and in Section 2.4 [Insertion], page 7. There are a couple of ways to customize these bindings.

`mmm-mode-prefix-key` [User Option]

The value of this variable (default is `C-c %`) should be a key sequence to use as the prefix for the MMM Mode keymap. Minor modes typically use `C-c` followed by a punctuation character, but you can change it to any user-available key sequence. To have an effect, this variable should be set before MMM Mode is loaded.

`mmm-use-old-command-keys` [User Option]

When this variable is `nil`, MMM Mode commands use the control modifier and insertion keys no modifier. Any other value switches the two, so that `mmm-parse-buffer`, for example, is bound to `C-c % b`, while perl-section insertion in the Mason class is bound to `C-c % C-p`. This variable should be set before MMM Mode is loaded to have an effect.

When MMM is loaded, it uses the value of `mmm-use-old-command-keys` to set the values of the variables `mmm-command-modifiers` and `mmm-insert-modifiers`, so if you prefer you can set these variables instead. They should each be a list of key modifiers, such as `(control)` or `()`. The Meta modifier is used in some of the command and insertion keys, so it should not be used, and the Shift modifier is not particularly portable between Emacsen—if it works for you, feel free to use it. Other modifiers, such as Hyper and Super, are not universally available, but are valid when present.

3.5 Changing Saved Local Variables

A lot of the functionality of MMM Mode—that which makes the major mode appear to change—is implemented by saving and restoring the values of local variables, or pseudo-variables. You can customize what variables are saved, and how, with the following variable.

`mmm-save-local-variables` [Variable]

At its simplest, this is a list each of whose elements is a buffer-local variable whose value is saved and restored for each major mode. Each elements can also, however, be a list whose first element is the variable symbol and whose subsequent elements specify how and where the variable is to be saved. The second element of the list, if present, should be one of the symbols `global`, `buffer`, or `region`. If not present, the default value is `global`. The third element, if present, should be a list of major mode symbols in which to save the variable. In the list form, the variable symbol itself can be replaced with a cons cell of two functions, one to get the value and one to set the value. This is called a “pseudo-variable”.

Globally saved variables are the same in all (MMM-controlled) buffers and submode regions of each major mode listed in the third argument, or all major modes if it is `t` or not present. Buffer-saved variables are the same in all submode regions of a given major mode in each buffer, and region-saved variables can be different for each submode region.

Pseudo-variables are used, for example, to save and restore the syntax table (see Section "Syntax" in *The Emacs Manual*) and mode keymaps (see Section "Keymaps" in *The Emacs Manual*).

3.6 Changing the Supplied Submode Classes

If you need to use MMM with a syntax for which a submode class is not supplied, and you have some facility with Emacs Lisp, you can write your own; see Chapter 5 [Writing Classes], page 20. However, sometimes you will only want to make a slight change to one of the supplied submode classes. You can do this, after that class is loaded, with the following functions.

`mmm-set-class-parameter` *class param value* [Function]
: Set the value of the keyword parameter *param* of the submode class *class* to *value*. See Chapter 5 [Writing Classes], page 20, for an explanation of the meaning of each keyword parameter. This creates a new parameter if one is not already present in the class.

`mmm-get-class-parameter` *class param* [Function]
: Get the value of the keyword parameter *param* for the submode class *class*. Returns `nil` if there is no such parameter.

3.7 Hooks Provided by MMM Mode

MMM Mode defines several hook variables (see Section "Hooks" in *The Emacs Manual*) which are run at different times. The most often used is `mmm-major-mode-hook` which is described in Section 2.7.1 [Major Mode Hook], page 9, but there are a couple others.

`mmm-mode-hook` [Variable]
: This normal hook is run whenever MMM Mode is enabled in a buffer.

`mmm-`*major-mode*`-hook` [Variable]
: This is actually a whole set of hook variables, a different one for every major mode. Whenever MMM Mode is enabled in a buffer, the corresponding hook variable for the dominant major mode is run.

`mmm-`*submode*`-submode-hook` [Variable]
: Again, this is a set of one hook variable per major mode. These hooks are run whenever a submode region of the corresponding major mode is created in any buffer, with point at the start of the new submode region.

`mmm-`*class*`-class-hook` [Variable]
: This is a set of one hook variable per submode class. These hooks are run when a submode class is first applied to a given buffer.

Submode classes also have a `:creation-hook` parameter which should be a function to run whenever a submode region is created with that class, with point at the beginning of the submode region. This can be set for supplied submode classes with `mmm-set-class-parameter`; Section 3.6 [Changing Classes], page 13.

4 Supplied Submode Classes

This chapter describes the submode classes that are supplied with MMM Mode.

4.1 Mason: Perl in HTML

Mason is a syntax to embed Perl code in HTML and other documents. See `http://www.masonhq.com` for more information. The submode class for Mason components is called 'mason' and is loaded on demand from 'mmm-mason.el'. The current Mason class is intended to correctly recognize all syntax valid in Mason 0.896. There are insertion keys for most of the available syntax; use `mmm-insertion-help` (*C-c % h* by default) with Mason on to get a list.

If you want to have mason submodes automatically in all Mason files, you can use automatic mode and filename associations; the details depend on what you call your Mason components and what major mode you use. See Section 2.3.2 [Mode-Ext Classes], page 6. If you use an extension for your Mason files that emacs does not automatically place in your preferred HTML Mode, you will probably want to associate that extension with your HTML Mode as well; Section "Choosing Modes" in *The Emacs Manual*. This also goes for "special" Mason files such as autohandlers and dhandlers.

The Perl mode used is controlled by the user: See Section 3.2 [Preferred Modes], page 11. The default is to use CPerl mode, if present. Unfortunately, there are also certain problems with CPerl mode in submode regions. (Not to say that the original perl-mode would do any better—it hasn't been much tried.) First of all, the first line of a Perl section is usually indented as if it were a continuation line. A fix for this is to start with a semicolon on the first line. The insertion key commands do this whenever the Mason syntax allows it.

```
<%perl>;
print $var;
</%perl>
```

In addition, some users have reported that the CPerl indentation sometimes does not work. This problem has not yet been tracked down, however, and more data about when it happens would be helpful.

Some people have reported problems using PSGML with Mason. Adding the following line to a `.emacs` file should suffice to turn PSGML off and cause emacs to use a simpler HTML mode:

```
(autoload 'html-mode "sgml-mode" "HTML Mode" t)
```

Earlier versions of PSGML may require instead the following fix:

```
(delete '("\\.html$" . sgml-html-mode) auto-mode-alist)
(delete '("\\.shtml$" . sgml-html-mode) auto-mode-alist)
```

Other users report using PSGML with Mason and MMM Mode without difficulty. If you don't have problems and want to use PSGML, you may need to replace `html-mode` in the suggested code with `sgml-html-mode`. (Depending on your version of PSGML, this may not be necessary.) Similarly, if you are using XEmacs and want to use the alternate HTML mode `hm--html-mode`, replace `html-mode` with that symbol.

One problem that crops up when using PSGML with Mason is that even ignoring the special tags and Perl code (which, as I've said, haven't caused me any problems), Mason

components often are not a complete SGML document. For instance, my autohandlers often say

```
<body>
  <% $m->call_next %>
</body>
```

in which case the actual components contain no doctype declaration, `<html>`, `<head>`, or `<body>`, confusing PSGML. One solution I've found is to use the variable `sgml-parent-document` in such incomplete components; try, for example, these lines at the end of a component.

```
%# Local Variables:
%# sgml-parent-document: ("autohandler" "body" nil ("body"))
%# sgml-doctype: "/top/level/autohandler"
%# End:
```

This tells PSGML that the current file is a sub-document of the file `autohandler` and is included inside a `<body>` tag, thus alleviating its confusion.

4.2 Elisp in a Local Variables List

Emacs allows the author of a file to specify major and minor modes to be used while editing that file, as well as specifying values for other local Elisp variables, with a File Variables list. See Section "File Variables" in *The Emacs Manual*. Since file variables values are Elisp objects (and with the `eval` special "variable", they are forms to be evaluated), one might want to edit them in `emacs-lisp-mode`. The submode class `file-variables` allows this, and is suitable for turning on in a given file with `mmm-classes`, or in all files with `mmm-global-classes`.

4.3 Here-documents

One of the long-time standard syntaxes for outputting large amounts of code (or text, or HTML, or whatever) from a script (notably shell scripts and Perl scripts) is the here-document syntax:

```
print <<END_HTML;
<html>
  <head>
    <title>Test Page</title>
  </head>
  <body>
END_HTML
```

The `here-doc` submode class recognizes this syntax, and can even guess the correct submode to use in many cases. For instance, it would put the above example in `html-mode`, noticing the string '`HTML`' in the name of the here-document. If you use less than evocative here-document names, or if the submode is recognized incorrectly for any other reason, you can tell it explicitly what submode to use.

`mmm-here-doc-mode-alist` [User Option]

The value of this variable should be an alist, each element a cons pair associating a regular expression to a submode symbol. Whenever a here-document name matches

one of these regexps, the corresponding submode is applied. For example, if this variable contains the element `("CODE" . cc-mode)`, then any here-document whose name contains the string '`CODE`' will be put in `cc-mode`. The value of this variable overrides any guessing that the `here-doc` submode class would do otherwise.

4.4 Javascript in HTML

The submode class `html-js` allows for embedding Javascript code in HTML documents. It recognizes both this syntax:

```
<script language="Javascript">
function foo(...) {
   ...
}
</script>
```

and this syntax:

```
<input type="button" onClick="validate();">
```

The mode used for Javascript regions is controlled by the user; See Section 3.2 [Preferred Modes], page 11.

4.5 CSS embedded in HTML

CSS (Cascading Style Sheets) can also be embedded in HTML. The `embedded-css` submode class recognizes this syntax:

```
<style>
h1 {
   ...
}
</style>
```

It uses `css-mode` if present, `c++-mode` otherwise. This can be customized: See Section 3.2 [Preferred Modes], page 11.

4.6 Embperl: More Perl in HTML

Embperl is another syntax for embedding Perl in HTML. See `http://perl.apache.org/embperl` for more information. The `embperl` submode class recognizes most if not all of the Embperl embedding syntax. Its Perl mode is also controllable by the user; See Section 3.2 [Preferred Modes], page 11.

4.7 ePerl: General Perl Embedding

Yet another syntax for embedding Perl is called ePerl. See `http://www.engelschall.com/sw/eperl/` for more information. The `eperl` submode class handles this syntax, using the Perl mode specified by the user; See Section 3.2 [Preferred Modes], page 11.

4.8 JSP: Java Embedded in HTML

JSP (Java Server Pages) is a syntax for embedding Java code in HTML. The submode class `jsp` handles this syntax, using a Java mode specified by the user; See Section 3.2 [Preferred Modes], page 11. The default is `jde-mode` if present, otherwise `java-mode`.

4.9 RPM Spec Files

`mmm-rpm.el` contains the definition of an MMM Mode submode class for editing shell script sections within RPM (Redhat Package Manager) spec files. It is recommended for use in combination with `rpm-spec-mode.el` by Stig Bjrlykke <stigb@tihlde.hist.no> and Steve Sanbeg <sanbeg@dset.com> (`http://www.xemacs.org/~stigb/rpm-spec-mode.el`).

Suggested setup code:

```
(add-to-list 'mmm-mode-ext-classes-alist
             '(rpm-spec-mode "\\.spec\\'" rpm-sh))
```

Thanks to Marcus Harnisch <Marcus.Harnisch@gmx.net> for contributing this submode class.

4.10 Noweb literate programming

`mmm-noweb.el` contains the definition of an MMM Mode submode class for editing Noweb documents. Most Noweb documents use \LaTeX for the documentation chunks. Code chunks in Noweb are document-specific, and the mode may be set with a local variable setting in the document. The variable *mmm-noweb-code-mode* controls the global code chunk mode. Since Noweb files may have many languages in their code chunks, this mode also allows setting the mode by specifying a mode in the first line or two of a code chunk, using the normal Emacs first-line mode setting syntax. Note that this first-line mode setting only matches a single word for the mode name, and does not support the variable name setting of the generalized first file line syntax.

```
% -*- mode: latex; mmm-noweb-code-mode: c++; -*-
% First chunk delimiter!
@
\noweboptions{smallcode}

\title{Sample Noweb File}
\author{Joe Kelsey\\
\nwanchorto{mailto:bozo@bozo.bozo}{\tt bozo@bozo.bozo}}
\maketitle

@
\section{Introduction}
Normal noweb documentation for the required [[*]] chunk.
<<*>>=
// C++ mode here!
// We might list the program here, or simply included chunks.
<<myfile.cc>>
@ %def myfile.cc
```

```
@
\section{[[myfile.cc]]}
This is [[myfile.cc]].  MMM noweb-mode understands code quotes in
documentation.
<<myfile.cc>>=
// This section is indented separately from previous.
@

@
\section{A Perl Chunk}
We need a Perl chunk.
<<myfile.pl>>=
#!/usr/bin/perl
# -*- perl -*-
# Each differently named chunk is flowed separately.
@

\section{Finish [[myfile.cc]]}
When we resume a previously defined chunk, they are indented together.
<<myfile.cc>>=
// Pick up where we left off...
@
```

The quoted code chunks inside documentation chunks are given the mode found in the variable *mmm-noweb-quote-mode*, if set, or the value in *mmm-noweb-code-mode* otherwise. Also, each quoted chunk is set to have a unique name to prevent them from being indented as a unit.

Suggested setup code:

```
(mmm-add-mode-ext-class 'latex-mode "\\.nw\\'" 'noweb)
(add-to-list 'auto-mode-alist '("\\.nw\\'" . latex-mode))
```

In mmm-noweb buffers, each differently-named code chunk has a different `:name`, allowing all chunks with the same name to get indented together.

This mode also supplies special paragraph filling operations for use in documentation areas of the buffer. From a primary-mode (`latex-mode, , emacs`) region, pressing *C-c % C-q* will mark all submode regions with word syntax (`mmm-word-other-regions`), fill the current paragraph (`(fill-paragraph justify)`), and remove the syntax markings (`mmm-undo-syntax-other-regions`).

Thanks to Joe Kelsey <joe@zircon.seattle.wa.us> for contributing this class.

5 Writing Submode Classes

Sometimes (perhaps often) you may want to use MMM with a syntax for which it is suited, but for which no submode is supplied. In such cases you may have to write your own submode class. This chapter briefly describes how to write a submode class, from the basic to the advanced, with examples.

5.1 Writing Basic Submode Classes

Writing a submode class can become rather complex, if the syntax to match is complicated and you want to take advantage of some of MMM Mode's extra features. But a simple submode class is not particularly difficult to write. This section describes the basics of writing submode classes.

Submode classes are stored in the variable `mmm-classes-alist`. Each element of this list represents a single submode class. For convenience, the function `mmm-add-classes` takes a list of submode classes and adds them all to this alist. Each class is represented by a list containing the class name—a symbol such as `mason` or `html-js`—followed by pairs of keywords and arguments called a *class specifier*. For example, consider the specifier for the submode class `embedded-css`:

```
(mmm-add-classes
 '((embedded-css
    :submode css
    :face mmm-declaration-submode-face
    :front "<style[^>]*>"
    :back "</style>")))
```

The name of the submode is `embedded-css`, the first element of the list. The rest of the list consists of pairs of keywords (symbols beginning with a colon) such as `:submode` and `:front`, and arguments, such as `css` and `"<style[^>]*>"`. It is the keywords and arguments that specify how the submode works. The order of keywords is not important; all that matters is the arguments that follow them.

The three most important keywords are `:submode`, `:front`, and `:back`. The argument following `:submode` names the major mode to use in submode regions. It can be either a symbol naming a major mode, such as `text-mode` or `c++-mode`, or a symbol to look up in `mmm-major-mode-preferences` (see Section 3.2 [Preferred Modes], page 11) such as `css`, as in this case.

The arguments following `:front` and `:back` are regular expressions (see Section "Regexps" in *The Emacs Manual*) that should match the delimiter strings which begin and end the submode regions. In our example, CSS regions begin with a '`<style>`' tag, possibly with parameters, and end with a '`</style>`' tag.

The argument following `:face` specifies the face (background color) to use when `mmm-submode-decoration-level` is 2 (high coloring). See Section 3.1 [Region Coloring], page 10, for a list of canonical available faces.

There are many more possible keywords arguments. In the following sections, we will examine each of them and their uses in writing submode classes.

5.2 Matching Paired Delimiters

A simple pair of regular expressions does not always suffice to exactly specify the beginning and end of submode regions correctly. For this reason, there are several other possible keyword/argument pairs which influence the matching process.

Many submode regions are marked by paired delimiters. For example, the tags used by Mason (see Section 4.1 [Mason], page 15) include '`<%init>...</%init>`' and '`<%args>...</%args>`'. It would be possible to write a separate submode class for each type of region, but there is an easier way: the keyword argument `:save-matches`. If supplied and non-nil, it causes the regular expression `:back`, before being searched for, to be formatted by replacing all strings of the form '`~`*N*' (where *N* is an integer) with the corresponding numbered subexpression of the match for `:front`. As an example, here is an excerpt from the `here-doc` submode class. See Section 4.3 [Here-documents], page 16, for more information about this submode.

```
:front "<<\\([a-zA-Z0-9_-]+\\)"
:back "^~1$"
:save-matches 1
```

The regular expression for `:front` matches '`<<`' followed by a string of one or more alphanumeric characters, underscores, and dashes. The latter string, which happens to be the name of the here-document, is saved as the first subexpression, since it is surrounded by '`\(...\)`'. Then, because the value of `:save-matches` is present and non-nil, the string '`~1`' is replaced in the value of `:back` by the name of the here-document, thus creating a regular expression to match the correct ending delimiter.

5.3 Placing Submode Regions Precisely

Normally, a submode region begins immediately after the end of the string matching the `:front` regular expression and ends immediately before the beginning of the string matching the `:back` regular expression. This can be changed with the keywords `:include-front` and `:include-back`. If their arguments are `nil`, or they do not appear, the default behavior is unchanged. But if the argument of `:include-front` (respectively, `:include-back`) is non-nil, the submode region will begin (respectively, end) immediately before (respectively, after) the string matching the `:front` (respectively, `:back`) regular expression. In other words, these keywords specify whether or not the delimiter strings are *included* in the submode region.

When `:front` and `:back` are regexps, the delimiter is normally considered to be the entire matched region. This can be changed using the `:front-match` and `:back-match` keywords. The values of the keywords is a number specifying the submatch. This defaults to zero (specifying the whole regexp).

Two more keywords which affect the placement of the region `:front-offset` and `:back-offset`, which both take integers as arguments. The argument of `:front-offset` (respectively, `:back-offset`) gives the distance in characters from the beginning (respectively, ending) location specified so far, to the actual point where the submode region begins (respectively, ends). For example, if `:include-front` is nil or unsupplied and `:front-offset` is 2, the submode region will begin two characters after the end of the match for `:front`, and if `:include-back` is non-nil and `:back-offset` is -1, the region will end one character before the end of the match for `:back`.

In addition to integers, the arguments of `:front-offset` and `:back-offset` can be functions which are invoked to move the point from the position specified by the matches and inclusions to the correct beginning or end of the submode region, or lists whose elements are either functions or numbers and whose effects are applied in sequence. To help disentangle these options, here is another excerpt from the `here-doc` submode class:

```
:front "<<\\([a-zA-Z0-9_-]+\\)"
:front-offset (end-of-line 1)
:back "^~1$"
:save-matches 1
```

Here the value of `:front-offset` is the list `(end-of-line 1)`, meaning that from the end of the match for `:front`, go to the end of the line, and then one more character forward (thus to the beginning of the next line), and begin the submode region there. This coincides with the normal behavior of here-documents: they begin on the following line and go until the ending flag.

If the `:back` should not be able to start a new submode region, set the `:end-not-begin` keyword to non-nil.

5.4 Defining Groups of Submodes

Sometimes more than one submode class is required to accurately reflect the behavior of a single type of syntax. For example, Mason has three very different types of Perl regions: blocks bounded by matched tags such as '`<%perl>...</%perl>`', inline output expressions bounded by '`<%...%>`', and single lines of code which simply begin with a '`%`' character. In cases like these, it is possible to specify an "umbrella" class, to turn all these classes on or off together.

`mmm-add-group` *group classes* [Function]

The submode classes *classes*, which should be a list of lists, similar to what might be passed to `mmm-add-classes`, are added just as by that function. Furthermore, another class named *group* is added, which encompasses all the classes in *classes*.

Technically, an group class is specified with a `:classes` keyword argument, and the subsidiary classes are given a non-nil `:private` keyword argument to make them invisible. But in general, all you should ever need to know is how to invoke the function above.

`mmm-add-to-group` *group classes* [Function]

Adds a list of classes to an already existing group. This can be used, for instance, to add a new quoting definition to *html-js* using this example to add the quote characters "%=%":

```
(mmm-add-to-group 'html-js '((js-html
     :submode javascript
     :face mmm-code-submode-face
     :front "%=%"
     :back "%=%"
     :end-not-begin t)))
```

5.5 Calculating the Correct Submode

In most cases, the author of a submode class will know in advance what major mode to use, such as `text-mode` or `c++-mode`. If there are multiple possible modes that the user might desire, then `mmm-major-mode-preferences` should be used (see Section 3.2 [Preferred Modes], page 11). The function `mmm-set-major-mode-preferences` can be used, with a third argument, to ensure than the mode is present.

In some cases, however, the author has no way of knowing in advance even what language the submode region will be in. The `here-doc` class is one of these. In such cases, instead of the `:submode` keyword, the `:match-submode` keyword must be used. Its argument should be a function, probably written by the author of the submode class, which calculates what major mode each region should use.

It is invoked immediately after a match is found for `:front`, and is passed one argument: a string representing the front delimiter. Normally this string is simply whatever was matched by `:front`, but this can be changed with the keyword `:front-form` (see Section 5.10 [Delimiters], page 26). The function should then return a symbol that would be a valid argument to `:submode`: either the name of a mode, or that of a language to look up a preferred mode. If it detects an invalid match—for example, the user has specified a mode which is not available—it should `(signal 'mmm-no-matching-submode nil)`.

Since here-documents can contain code in any language, the `here-doc` submode class uses `:match-submode` rather than `:submode`. The function it uses is `mmm-here-doc-get-mode`, defined in `mmm-sample.el`, which inspects the name of the here-document for flags indicating the proper mode. For example, this code should probably be in `perl-mode` (or `cperl-mode`):

```
print <<PERL;
s/foo/bar/g;
PERL
```

This function is also a good example of proper elisp hygiene: when writing accessory functions for a submode class, they should usually be prefixed with '`mmm-`' followed by the name of the submode class, to avoid namespace conflicts.

5.6 Calculating the Correct Highlight Face

As explained in Section 5.1 [Basic Classes], page 20, the keyword `:face` should be used to specify which of the standard submode faces (see Section 3.1 [Region Coloring], page 10) a submode region should be highlighted with under high decoration. However, sometimes the function of a region can depend on the form of the delimiters as well. In this case, a more flexible alternative to `:face` is `:match-face`. Its value can be a function, which is called with one argument—the form of the front delimiter, as with `:match-submode`—and should return the face to use. A more common value for `:match-face` is an association list, a list of pairs `(`*`delim`* `.` *`face`*`)`, each specifying that if the delimiter is *delim*, the corresponding region should be highlighted with *face*. For example, here is an excerpt from the `embperl` submode class:

```
:submode perl
:front "\\[\\([-\\+!\\*\\$]\\)"
:back "~1\\]"
```

```
:save-matches 1
:match-face (("[+" . mmm-output-submode-face)
             ("[-" . mmm-code-submode-face)
             ("[!" . mmm-init-submode-face)
             ("[*" . mmm-code-submode-face)
             ("[$" . mmm-special-submode-face))
```

Thus, regions beginning with '`[+`' are highlighted as output expressions, which they are, while '`[-`' and '`[*`' regions are highlighted as simple executed code, and so on. Note that *mmm-submode-decoration-level* must be set to 2 (high decoration) for different faces to be displayed.

5.7 Specifying Insertion Commands

As described in Section 2.4 [Insertion], page 7, submode classes can specify key sequences which automatically insert submode regions, with delimiters already in place. This is done by the keyword argument `:insert`. Its value should be a list, each element of which specifies a single insertion key sequence. As an example, consider the following insertion key sequence specifier, from the `embperl` submode class:

```
(?p embperl "Region Type (Character): "
    @ "[" str @ " " _ " " @ str "]" @)
```

As you can see, the specifier is a list. The first element of the list is the character '`p`'. (The question mark tells Emacs that this is a character object, not a one-character symbol.) In general, the first element can be any key, including both characters such as '`?p`' and function keys such as '`return`'. It can also be a dotted pair in which the first element is a modifier symbol such as `meta`, and the second is a character or function key. The use of any other modifier than meta is discouraged, as 'mmm-insert-modifiers' is sometimes set to \(control), and other modifiers are not very portable. The second element is a symbol identifying this key sequence. The third element is a prompt string which is used to ask the user for input when this key sequence is invoked. If it is nil, the user is not prompted.

The rest of the list specifies the actual text to be inserted, where the submode region and delimiters should be, and where the point should end up. (Actually, this string is simply passed to `skeleton-insert`; see the documentation string of that function for more details on the permissible elements of such a skeleton.) Strings and variable names are inserted and interpolated. The value entered by the user when prompted, if any, is available in the variable `str`. The final location of the point (or the text around which the region is to be wrapped) is marked with a single underscore '`_`'. Finally, the @-signs mark the delimiters and submode regions. There should be four @-signs: one at the beginning of the front delimiter, one at the beginning of the submode region, one at the end of the submode region, and one at the end of the back delimiter.

The above key sequence, bound by default to *C-c % p*, always prompts the user for the type of region to insert. It can also be convenient to have separate key sequences for each type of region to be inserted, such as *C-c % +* for '`[+...+]`' regions, *C-c % -* for '`[-...-]`' regions, and so on. So that the whole skeleton doesn't have to be written out half a dozen times, there is a shortcut syntax, as follows:

```
(?+ embperl+ ?p . "+")
```

If the key sequence specification is a dotted list with four elements, as this example is, it means to use the skeleton defined for the key sequence given as the third element (`?p`), but to pass it the fourth (dotted) element (`"+"`) as the 'str' variable; the user is not prompted.

5.8 Giving Names to Submode Regions for Grouping

Submode regions can be given "names" which are used for grouping. Names are always strings and are compared as strings. Regions with the same name are considered part of the same chunk of code. This is used by the syntax and fontification functions. Unnamed regions are not grouped with any others.

By default, regions are nameless, but with the `:match-name` keyword argument a name can be supplied. This argument must be a string or a function. If it is a function, it is passed a string representing the front delimiter found, and must return the name to use. If it is a string, it is used as-is for the name, unless `:save-name` has a non-nil value, in which case expressions such as '`~1`' are substituted with the corresponding matched subexpression from `:front`. This is the same as how `:back` is interpreted when `:save-matches` is non-nil.

As a special optimization for region insertion (see Section 5.7 [Insertion Commands], page 24), the argument `:skel-name` can be set to a non-nil value, in which case the insertion code will use the user-prompted string value as the region name, instead of going through the normal matching procedure.

5.9 Other Hooks into the Scanning Process

Sometimes, even the flexibility allowed by all the keyword arguments discussed so far is insufficient to correctly match submode regions. There are several other keyword arguments which accept custom functions to be invoked at various points in the MMM-ification process.

First of all, the arguments of `:front` and `:back`, in addition to regular expressions, can be themselves functions. Such functions should "act like" a regular expression search: they should start searching at point, take one argument as a limit for the search, and return its result by setting the match data (presumably by calling some regexp matching function).

This is rarely necessary, however, because often all that is needed is a simple regexp search, followed by some sort of verification. The keyword arguments `:front-verify` and `:back-verify`, if supplied, may be functions which are invoked after a match is found for `:front` or `:back`, respectively, and should inspect the match data (such as with `match-string`) and return non-nil if a submode region should be begun at this match, nil if this match should be ignored and the search continue after it.

The keyword argument `:creation-hook`, if supplied, should be a function that is invoked whenever a submode region of this class is created, with point at the beginning of the new region. This can be used, for example, to set local variables appropriately.

Finally, the entire MMM-ification process has a "back door" which allows class authors to take control of the entire thing. If the keyword argument `:handler` is supplied, it overrides any other processing and is called, and passed all other class keyword arguments, instead of `mmm-ify` to create submode regions. If you need to write a handler function, I suggest looking at the source for `mmm-ify` to get an idea of what must be done.

5.10 Controlling the Delimiter Regions and Forms

MMM also makes overlays for the delimiter regions, to keep track of their position and form. Normally, the front delimiter overlay starts at the beginning of the match for `:front` and ends at the beginning of the submode region overlay, while the back delimiter overlay starts at the end of the submode region overlay and ends at the end of the match for `:back`. You can supply offsets from these positions using the keyword arguments `:front-delim` and `:back-delim`, which take values of the same sort as `:front-offset` and `:back-offset`.

In addition, the delimiter regions can be in a major mode of their own. There are usually only two meaningful modes to use: the primary mode or a non-mode like fundamental-mode. These correspond to the following two situations:

- If the delimiter syntax which specifies the submode regions is something *added to* the syntax of the primary mode by a pre-interpreter, then the delimiter regions should be in a non-mode. This is the case, for example, with all server-side HTML script extensions, such as See Section 4.1 [Mason], page 15, See Section 4.6 [Embperl], page 17, and See Section 4.7 [ePerl], page 17. It is also the case for literate programming such as See Section 4.10 [Noweb], page 18. This is the default behavior. The non-mode used is controlled by the variable `mmm-delimiter-mode`, which defaults to fundamental-mode.
- If, on the other hand, the delimiter syntax and inclusion of different modes is an *intrinsic part* of the primary mode, then the delimiter regions should remain in the primary mode. This is the case, for example, with See Section 4.5 [Embedded CSS], page 17, and See Section 4.4 [Javascript], page 17, since the `<style>` and `<script>` tags are perfectly valid HTML. In this case, you should give the keyword parameter `:delimiter-mode` with a value of `nil`, meaning to use the primary mode.

The keyword parameter `:delimiter-mode` can be given any major mode as an argument, but the above two situations should cover the vast majority of cases.

The delimiter regions can also be highlighted, if you wish. The keyword parameters `:front-face` and `:back-face` may be faces specifying how to highlight these regions under high decoration. Under low decoration, the value of the variable `mmm-delimiter-face` is used (by default, nothing), and of course under no decoration there is no coloring.

Finally, for each submode region overlay, MMM Mode stores the "form" of the front and back delimiters, which are regular expressions that match the delimiters. At present these are not used for much, but in the future they may be used to help with automatic updating of regions as you type. Normally, the form stored is the result of evaluating the expression `(regexp-quote (match-string 0))` after each match is found.

You can customize this with the keyword argument `:front-form` (respectively, `:back-form`). If it is a string, it is used verbatim for the front (respectively, back) form. If it is a function, that function is called and should inspect the match data and return the regular expression to use as the form.

In addition, the form itself can be set to a function, by giving a one-element list containing only that function as the argument to `:front-form` or `:back-form`. Such a function should take 1-2 arguments. The first argument is the overlay to match the delimiter for. If the second is non-nil, it means to insert the delimiter and adjust the overlay; if nil it means to match the delimiter and return the result in the match data.

5.11 Miscellaneous Other Keyword Arguments

You can specify whether delimiter searches should be case-sensitive with the keyword argument `:case-fold-search`. It defaults to `t`, meaning that case should be ignored. See the documentation for the variable `case-fold-search`.

6 Indices

6.1 Concept Index

6.2 Function and Variable Index

6.3 Keystroke Index