Programming Language Support — Emacs Lisp

Operation	Keystroke	Function	Notes
Emacs Lisp Editing	• To edit Emacs Lisp code, the Emacs Lisp major mode is normally used. • Some of the key bindings listed in this table are available from all modes or some other modes (like the PEL key bindings highlighted with light green). Some other are context sensitive and only available for the Emacs Lisp major mode (like the PEL **<f12>** or **<M-f12>** key prefixes, which are highlighted in darker green). Those can also be accessed via the **<f11> SPC l** prefix. These are not all written in the following rows to save space. • Some of the commands are meant to be used regardless of the mode, but were documented in this table because they are available everywhere, are essentially controlling or explicitly using the Emacs Lisp engine or environment in such a way so the user must be aware of Emacs Lisp and the available commands. These bindings **coloured in violet.**		
Customize PEL EmacsLisp Support (See also: ∑ Customize)	• **<f11> <f1> SPC l** • **<f12> <f1>**	(**pel-cfg-pkg-elisp** &optional OTHER-WINDOW)	Customize PEL EmacsLisp support. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window and open emacs lisp related customization groups as well. • The **<f12> <f1>** binding is available when point is in a buffer visiting a EmacsLisp file.
Extra Modes	The following commands can be used to activate or toggle useful modes for Emacs Lisp editing, specially for helping dealing with parenthesis: • show-paren-mode, which highlights the parens that matches the one before or after point. • ParInfer mode (with either ParInfer Indent Mode or Parinfer Paren Mode) where the parenthesis or indentation is automatically inferred from the other. • rainbow delimiters mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off (see also: ∑ Highlight)	• **<f12> M-9** • **<M-f12> M-9** • **<f11> SPC l M-9** • **<f11> b h (**	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With a prefix argument ARG, enable Show Paren mode if ARG is positive, and disable it otherwise. • Show Paren mode is a global minor mode. When enabled, any matching parenthesis is highlighted in 'show-paren-style' after 'show-paren-delay' seconds of Emacs idle time.
Enable/Disable coloured highlight of nested blocks (),{},[] (see also: ∑ Highlight)	• **<f12> M-r** • **<M-f12> M-r** • **<f11> SPC l m R** • **<f11> b h R**	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** 📦 **Requires:** rainbow-delimiters.el PEL activates this when the **pel-use-rainbow-delimiters** customize variable is set to **t**.
Toggle Lisp Defined Symbol Highlight	• **<f12> M-d** • **<M-f12> M-d** • **<f11> SPC l M-d**	(**highlight-defined-mode** &optional ARG)	Minor mode for highlighting known Emacs Lisp functions and variables. • Toggle highlight defined mode on or off. With a prefix argument ARG, enable highlight defined mode if ARG is positive, and disable it otherwise. Mainly useful while editing Emacs Lisp source code files. 📦 **Requires:** highlight-defined.el PEL activates this when the **pel-use-highlight-defined** customize variable is set to **t**.
Toggle ParInfer mode on/off	• **<f12> M-i** • **<M-f12> M-i** • **<f11> SPC l M-i**	(**parinfer-mode** &optional ARG)	Toggle use of the ParInfer mode. In this mode parenthesis depth or indentation is automatically inferred. ⚠ Current implementation of ParInfer does not support hard tabs for indentation. It untabifies and replace them by spaces. ☛📦 Requires the parinfer package. ☛ PEL activates this when the **pel-use-parinfer** customize variable is set to **t**.
Toggle between ParInfer Indent Mode and Paren Mode	• **<f12> M-I** • **<M-f12> M-I** • **<f11> SPC l M-I**	(**parinfer-toggle-mode**)	Switch ParInfer mode between Indent Mode and Paren Mode. • ⚠ Note that if the ParInfer mode is not active yet, and it enters ParInfer Indent Mode, the function checks the style of the current buffer and proceed with changing the format after prompting when it finds code that does not conform to the promoted style. The 2 ParInfer modes are: 1. ParInfer **Indent Mode**: • Gives full control of indentation, while ParInfer corrects parens. • Disables the rainbow-delimiter-mode if used, to show closing parens in light gray since they can change as code indentation is changed. • ⚠ When changing to Indent Mode, ParInfer may correct the parentheses format if the code does not corresponds to the promoted style. 2. ParInfer **Paren Mode**: • Gives full control of parens, while ParInfer controls indentation. • Activates rainbow-delimiters-mode if available, showing matching parens in same colors. • 💡 Paren Mode can be used to fix incorrectly indented code before using Indent Mode. ☛📦 Requires the parinfer package. ☛ PEL activates this when the **pel-use-parinfer** customize variable is set to **t**.
Toggle between Lisp modes	• **<f12> M-l** • **<M-f12> M-l** • **<f11> SPC l M-l**	(**pel-toggle-lisp-modes**)	Toggle buffer's LISP mode: 'lisp-interaction-mode' <-> 'emacs-lisp-mode'. ☛ Useful if you want to use **C-j** to evaluate and print value of the sexp before point while editing an Emacs Lisp (.el) file: when editing .el file, Emacs is normally in emacs-lisp-mode where C-j is mapped to electric-newline-and-maybe-indent. Temporarily changing to lisp-interaction-mode maps **C-j** to eval-print-last-sexp.
Toggle semantic parser mode on/off	• **<f12> M-s** • **<M-f12> M-s** • **<f11> SPC l M-s**	(**semantic-mode** &optional ARG)	Toggle parser features (Semantic mode). • With a prefix argument ARG, enable Semantic mode if ARG is positive, and disable it otherwise. If called from Lisp, enable Semantic mode if ARG is omitted or nil. • In Semantic mode, Emacs parses the buffers you visit for their semantic content.
Emacs Lisp Evaluation	GNU Emacs is implemented in Emacs Lisp with low level code written in C. Some of the functions can be used interactively; these functions are called commands. Some of these commands are bound to a key or a combination of keys (called key bindings). ‣ This section shows the commands (and their key bindings) you can use to explicitly evaluate Emacs Lisp code. ‣ The bindings **coloured in violet** are always available, not only when point is inside a Emacs Lisp buffer (unless the keychord was rebound).		
Execute Emacs Command	**M-x** <command>	(**execute-extended-command** PREFIXARG &optional COMMAND-NAME TYPED)	Read a command name, then read the arguments and call the command. To pass a prefix argument to the command you are invoking, use a prefix argument. • From the prompt you can press **<tab>** to perform completion and to list the names of the Emacs commands available. • To see the list of available commands, type **M-x <tab> <tab>** then press **<tab>** again to scroll the (large) list. • To quit this command, type **C-g** or **<Esc> <Esc><Esc>**.
Read & eval mini buffer	**M-:**	(**eval-expression** EXP &optional INSERT-VALUE NO-TRUNCATE CHAR-PRINT-LIMIT)	Read a single Emacs Lisp expression in the mini buffer, evaluate it, and print the value in the echo area.
Eval sexp before cursor	**C-x C-e**	(**eval-last-sexp** EVAL-LAST-SEXP-ARG-INTERNAL)	Evaluate sexp before point; print value in the echo area. Interactively, with a non '-' prefix argument, print output into current buffer: ie: **C-u C-x C-e** prints output to the current buffer. ☛ Binding available in the Emacs-Lisp (editing .el files) and Lisp-Interaction (the *Scratch* buffer) mode.

Operation	Keystroke	Function	Notes
Evaluate Lisp-Expression (defun) at point	`C-M-x`	(**eval-defun** EDEBUG-IT)	Evaluate the top-level form containing point, or after point. • Not restricted to a defun, it supports all definition forms. • With a prefix argument (**C-u**), instrument the code for Edebug (see edebug section below). ☛ Binding available in the Emacs-Lisp (editing .el files) and Lisp-Interaction (the *Scratch* buffer) mode.
Evaluate Lisp S-expression before point	`C-j`	(**eval-print-last-sexp** &optional EVAL-LAST-SEXP-ARG-INTERNAL)	Evaluate sexp before point; print value into current buffer. • For example, use this inside the *Scratch* buffer: place the cursor after a slips expression and type **C-j** to evaluate the expression. Emacs evaluate & run the expression and prints the returned value. ⚠ The **C-j** binding is **only available in the Lisp-Interaction mode** (the default mode of the *Scratch* buffer but not the default mode for editing Emacs Lisp files. You can use **<f12> m L**, (**pel-toggle-lisp-modes**), to temporarily change mode and activate the binding in the .el file buffer.
Eval all Emacs Lisp expressions in the buffer	• **<f12> e b** • **<M-f12> e b** • **<f11> SPC l e b**	(**eval-buffer** &optional BUFFER PRINTFLAG FILENAME UNIBYTE DO-ALLOW-PRINT)	Execute the accessible portion of current buffer as Lisp code. • You can use **C-x n n** (narrowing) to limit the part of buffer to be evaluated. • This function preserves the position of point.
Load and eval Emacs Lisp file	• **<f12> e f** • **<M-f12> e f** • **<f11> SPC l e f**	(**load-file** FILE)	Load the Emacs Lisp file named FILE. • Emacs prompts for the .el or .el.gz file name.
Evaluate all Emacs Lisp expressions in region	• **<f12> e r** • **<M-f12> e r** • **<f11> SPC l e r**	(**eval-region** START END &optional PRINTFLAG READ-FUNCTION)	Execute the region as Lisp code. • This function preserves the position of point.
Emacs Lisp shell (See also: ∑ Shells)	**<f11> x i**	(**ielm**)	Open the Interactive Emacs Lisp Mode buffer where you can interactively evaluate Emacs Lisp expressions, a REPL for Emacs Lisp. • Switches to the buffer '***ielm***', or creates it if it does not exist.
Semantic Editing	Several of the commands for editing Common Lisp code are also available for other modes and are described in the tables describing the generic Emacs commands (the pages with a title that begin with the character '∑'). These commands are repeated here for convenience; their keystroke cell is filled with a pale yellow colour. Several of them are described, with code examples, in the Common Lisp Cookbook - Using Emacs as a Lisp IDE page: this also mostly applies to Emacs Lisp code.		
SemEd - Kill			
Kill next Lisp S-expression (See also: ∑ Cut & Paste)	• **C-M-k** • **<f11> -]**	(**kill-sexp** &optional ARG)	• No argument: kill the next sexp (or the current from the point forward). • With negative sign: kill the previous sexp (the sexp backward). • For example: **M- - C-M-k** kills the sexp backward. • With numeric argument: kill that many sexp in the direction identified by the sign of the argument.
Kill previous Lisp S-expression (See also: ∑ Cut & Paste)	• **C-M-⌦** • **<f11> - [**	(**backward-kill-sexp** &optional ARG)	Kill the sexp (balanced expression) preceding point. • With ARG, kill that many sexps before point. • Negative arg -N means kill N sexps after point. • This command assumes point is not in a string or comment. ⚠ Note: In some text (like The Common Lisp Cookbook - Using Emacs as a Lisp IDE) , the **C-M-<backspace>** keystroke is being described to kill the previous sexp. This key does not seem to be used anymore. This key chord is normally not accessible in terminal mode as it would map to **C-M-h** instead. The **C-M-⌦** binding only works in terminal mode. Since this key-chord is not the best match for the operation, use **M- - C-M-k** instead or use the PEL **<f11> - [**
Kill Lisp S-Expression at point (See also: ∑ Cut & Paste)	**<f11> - x**	(**pel-kill-sexp-at-point**)	Kill the S-Expression at point. The point must be at the opening parenthesis or just after the closing parenthesis.
Code Completion & Spell Checking	Code auto completion and spell checking is available for Emacs Lisp source code files. Spell checking should be restricted to comments and strings, and code completion available everywhere else.		

Complete a partially typed word or Emacs Lisp symbol
 	
(See also:
• ∑ auto-completion
• ∑ Spell Checking
)
	• **M-<tab>**
	• **C-M-i**
	• **C-.**
		(**completion-at-point**)	Perform completion on the text around point.
			The completion method is determined by '**completion-at-point-functions**'. For Emacs Lisp code this is normally (**tags-completion-at-point-function**) which uses the tag facility to identify the choices, shown in a completion buffer.
			☝ Interaction with Flyspell:
			  • The key binding is affected by Flyspell: when Flyspell mode is active (either for the entire file or just for comment and strings) then the key chord is bound to (**flyspell-auto-correct-word**) instead. However, when the command is issued inside code, then Flyspell invokes code completion function (completion-at-point) such that the completion of the code is done the way it would be normally.
			  • You can use **<f11> $ F** (flyspell-mode &optional ARG) to activate Flyspell or **<f11> $ p** (flyspell-prog-mode) to activate Flyspell but restrict it to spell check comment and strings. See the ∑ Spell Checking table for more information.

Enter/Leave Flyspell mode (See also: ∑ Spell Checking)	**<f11> $ F**	(**flyspell-mode** &optional ARG)	Toggles the use of Flyspell mode. • Mode line shows "Fly" when Flyspell mode is active. • Flyspell mode works like word processors; misspelled words are highlighted. • Use Flyspell Prog mode for code; Flyspell processes all text. • With a prefix argument ARG, enable Flyspell mode if ARG is positive, and disable it otherwise. • Flyspell mode is a buffer-local minor mode. When enabled, it spawns a single ispell/aspell process and checks each word. The default flyspell behavior is to highlight incorrect words. ☞ You should normally not activate Flyspell everywhere in an Emacs Lisp file. However, if you activate it only for comments and strings with the command shown below, and then if you want to disable it you will have to disable the Flyspell mode completely with this command.
Enter Flyspell Prog mode (See also: ∑ Spell Checking)	**<f11> $ p**	(**flyspell-prog-mode**)	Turn on Flyspell prog mode: turn on Flyspell but restricts it to comments and strings, do not spell check source code itself. Highlight misspellings only in comments or strings. ☝ If a hook activates Flyspell Prog mode, you won't need this command. ⚠ Note that the command always enables the mode, it does not toggle it. If you want to turn spell checking off, you must use the flyspell-mode command. To re-enable Flyspell Prog mode you then use this one.
SemEd - Mark			
mark function (See also: ∑ Marking)	**C-M-h**	(**mark-defun** &optional ALLOW-EXTEND)	Put mark at end of this defun, point at beginning. • The defun marked is the one that contains point or follows point. • With positive ARG, mark this and that many next defuns; with negative ARG, change the direction of marking. • If the mark is active, it marks the next or previous defun(s) after the one(s) already marked.

Operation	Keystroke	Function	Notes
mark sexp and balanced expressions (See Also: ∑ Marking)	• **Esc C-@** • **C-M-@** • **C-M-SPC** • **<f11> . x**	(**mark-sexp** &optional ARG ALLOW-EXTEND)	Set mark ARG sexps (and balanced expressions) from point. • The place mark goes is the same place C-M-f would move to with the same argument. • Interactively, if this command is repeated or (in Transient Mark mode) if the mark is active, it marks the next ARG sexps after the ones already marked. • This command assumes point is not in a string or comment.
Mark region by semantic unit, increase marked region on each invocation. **★Powerful command ★** (See also: ∑ Marking)	• **M-=** • **<f11> . =**	(**er/expand-region** ARG)	Increase selected region by semantic units. • With prefix argument expands the region that many times. • If prefix argument is negative calls 'er/contract-region'. • If prefix argument is 0 it resets point and mark to their state before calling 'er/expand-region' for the first time. This command is very powerful: the first time it's typed it selects a word, if you type it again it will expand the selection, and again, and again. The expansions follow the semantics of the current major mode: it is aware of the semantics of several programming languages. ☛ Once M-= is typed, you can quickly type the following single keys in sequence: • **=** to expand the region, • **-** to contract the region, • **0** to reset the operation. If you wait too long, then you have to use **M-=** again to continue the expansion, otherwise the region is de-activated. Note that you can also use the following key chords to control the contraction of the selected text without having to worry about time: • **M- M-=** to contract the region • **M-0 M-=** to reset the operation. • Also you can use the cursor keys to expand or contract the region and **C-x C-x** to exchange mark and point to expand the other side of the region with cursors. 📦 This requires the expand-region package. ☛ Under PEL, activated with ***pel-use-expand-region*** customize variable. ☛ The PEL package uses this command and key binding for it, a popular binding for this command is **C-=** but that key does not work in text terminal mode. The standard Emacs binding for **M-=** is normally **count-words-region** used for counting words in region, but PEL provides **<f11> c r** for that.
SemEd - Navigation	The following commands help navigate across code blocks and parentheses (S-expressions in Lisp).		
Find source code of function/variable at point	• **<f12> .** • **<M-f12> .** • **<f11> SPC l .**	(**pel-find-thing-at-point**)	Find source code of function or variable at point. • Open in current window unless a **C-u** prefix is supplied as IN-OTHER-WINDOW in which case it opens inside the other window. 🚧 The **M-.** key, part of the cross-reference support, is better for most purpose and it allows going back to the original location, which this one doe but only via the mark ring. This command might be removed. **TODO:** more investigation needed.
Move block backward (See Also: ∑ Navigation) (CLCB s1.lisp)	• **C-M-b** • **C-M-<left>**	(**backward-sexp** &optional ARG)	Move backward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move forward across N balanced expressions. This command assumes point is not in a string or comment. • **C-M-b** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<left>** : ☛ Shift marking works with this command. ❖ **C-M-<left>** does not work on Windows, but **H-<left>** works.
Move block forward (See Also: ∑ Navigation) (CLCB s1.lisp)	• **C-M-f** • **C-M-<right>**	(**forward-sexp** &optional ARG)	Move forward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move backward across N balanced expressions. This command assumes point is not in a string or comment. • **C-M-f** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<right>** : ☛ Shift marking works with this command. ❖ **C-M-<right>** does not work on Windows, but **H-<right>** does.
Up/inside sexp hierarchy (See Also: ∑ Navigation) (CLCB s1.lisp)	• **C-M-u** • **C-M-<up>**	(**backward-up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move backward out of one level of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. With ARG, do this that many times. A negative argument means move forward but still to a less deep spot. • **C-M-u** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<up>** : ☛ Shift marking works with this command. ❖ **C-M-<up>** does not work on Windows, but **H-<up>** does.
Down/inside sexp/block (See Also: ∑ Navigation) (CLCB s1.lisp)	• **C-M-d** • **C-M-<down>**	(**down-list** &optional ARG)	Move forward down one level of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do this that many times. A negative argument means move backward but still go down a level. • This command assumes point is not in a string or comment. • **C-M-d** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<down>** : ☛ Shift marking works with this command. ❖ **C-M-<down>** does not work on Windows, but **H-<down>** does.
Backward block/list (See Also: ∑ Navigation)	**C-M-p**	(**backward-list** &optional ARG)	Move backward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move forward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-p** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Forward block/list (See Also: ∑ Navigation)	**C-M-n**	(**forward-list** &optional ARG)	Move forward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move backward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-n** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Backward to beginning of defun (See Also: ∑ Navigation)	• **C-M-a** • **C-M-<home>** • **<f6> p** • **<f6> <up>** • **<f12> f p** • **<M-f12> f p** • **<f11> SPC l f p**	(**beginning-of-defun** &optional ARG)	Move backward to the beginning of a defun. • With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. ☛Shift marking is available in graphics mode, not in terminal mode (for **C-M-a** and **C-M-<home>**). However **<f6> p** and **<f6> <up>** handle Shift-marking fine in terminal mode. ⚠ This command moves to the beginning go the next function or of the same nesting level of the current location. It skips the functions and methods that are more deeply nested.

Operation	Keystroke	Function	Notes
Forward to end of defun (See Also: ∑ Navigation)	• **C-M-e** • **C-M-<end>** • **<f6> <right>**	(**end-of-defun** &optional ARG)	Move forward to next end of defun. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. ☛Shift marking is available in graphics mode, not in terminal mode (for **C-M-e** and **C-M-<end>**). However **<f6> <right>** handle Shift-marking fine in terminal mode. ⚠ This command moves to the end of the next **top-level** function or class. It skips the nested functions and methods.
Forward to start of next defun	• **<f6> n** • **<f6> <down>** • **<f12> f n** • **<M-f12> f n** • **<f11> SPC l f n**	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function definition. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`**. ☛Shift marking is available. ☝ This command complements what end-of-defun does. • It moves forward but not to the end of the function definition (like end-of-defun) but to the beginning of the function definition, which is often what users of other editors expect. • It handles nested functions or class methods in languages like Python and others.
Backward to end of previous define	**<f6> <left>**	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function definition. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`**. ☛Shift marking is available. ☝ This command complements this set of 4 commands. • 🚧 It handles most nested functions or class methods in languages like Python and others but not always. In some cases it does not move the point. Better logic is needed.
Move by sentences (See Also: ∑ Navigation)	The variable 'sentence-end' is a regular expression that matches ends of sentences. Useful in comments. In code it moves to the beginning or end of a definition form (defun, defmacro, etc…)		
To beginning of sentence	**M-a**	(**backward-sentence** &optional ARG)	Move backward to start of sentence. With arg, do it arg times. ☛ Shift marking works with this command.
To end of sentence	**M-e**	(**forward-sentence** &optional ARG)	Move forward to next end of sentence. With argument, repeat. With negative argument, move backward repeatedly to start of sentence. ☛ Shift marking works with this command.
Recentering	The following 2 command do **not** move point, but reposition the text in the current window.		
Position current line to window's Center / Bottom / Top . **Refresh screen.** (See Also: ∑ Navigation)	**C-l**	(**recenter-top-bottom** &optional ARG)	Without argument: moves the current line to window: center -> top -> bottom. • With arg: center -> bottom -> center -> top • With negative arg: bottom -> bottom -> center -> top • With numeric positive: move current line to window top position N, 0 := top window line • With negative numeric: move current line to bottom window position: -0 := bottom line
Reposition comment/ definition in full view (See Also: ∑ Navigation)	**C-M-l**	(**reposition-window** &optional ARG)	Attempts to make the current comment, current definition, or current block of code (or data) fully visible by scrolling the lines without changing the point. • Further invocations move it to the top of the window or toggle the visibility of comments that precede it (by scrolling the lines). ☛ This command helps when reviewing an area of Common Lisp code, allowing you to see the complete block of interest, and possibly, also the comments preceding it.
SemEd - Indenting	The indentation rules of Common Lisp code differ from the ones for Emacs Lisp. The indentation is controlled by a function bound to the Emacs variable ***lisp-indent-function*** . For Emacs Lisp the function to use is ***lisp-indent-function*** .		
Indent current line (or region)	**<tab>**	(**indent-for-tab-command** &optional ARG)	Indent the current line or region, or insert a tab, as appropriate. • This function either inserts a tab, or indents the current line, or performs symbol completion, depending on 'tab-always-indent'. The function called to actually indent the line or insert a tab is given by the variable 'indent-line-function'. • If a prefix argument is given, after this function indents the current line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, to reflect the current line's indentation. • In most major modes, if point was in the current line's indentation, it is moved to the first non-whitespace character after indenting; otherwise it stays at the same position relative to the text. • If 'transient-mark-mode' is turned on and the region is active, this function instead calls 'indent-region'. In this case, any prefix argument is ignored.
Indent lines of list after point (See also: ∑ Indentation)	**C-M-q**	(**indent-pp-sexp** &optional ARG)	Indent each line of the list starting just after point, or pretty-print it. • A prefix argument (**C-u**) specifies pretty-printing. Pretty-printing essentially uses more lines as it places the beginning of each list on a new line.
Untabify and re-indent complete buffer with ParInfer	• **<f12> i** • **<M-f12> i** • **<f11> SPC l i**	(**parinfer-auto-fix**)	Untabify whole buffer then reindent whole buffer. ☛📦 Requires the parinfer package. ☛ PEL activates this when the **pel-use-parinfer** customize variable is set to **t**.
SemEd - Parentheses	The commands below are used to help dealing with the parentheses (along with the semantic editing navigation commands listed above). Note that when the ParInfer mode is used, these are not required: in that mode you can type the parentheses characters and that will perform the same.		
Insert Parentheses (See also: 𝔓𝔢 Common Lisp, CLCB s4.lisp)	**M-(**	(**insert-parentheses** &optional ARG)	Enclose following ARG sexps in parentheses. • Leave point after open-paren. • A negative ARG encloses the preceding ARG sexps instead. • No argument is equivalent to zero: just insert '()' and leave point between. • If 'parens-require-spaces' is non-nil, this command also inserts a space before and after, depending on the surrounding characters. For Lisp it's best to have this set to non-nil. • If region is active, insert enclosing characters at region boundaries. • This command assumes point is not in a string or comment.
Move past close ')' and reindent (See also: 𝔓𝔢 Common Lisp)	**M-)**	(**move-past-close-and-reindent**)	Move past next ')', delete indentation before it, then indent after it. • Used to add another entry in the parent list.
Disabling/Enabling Commands	Some Emacs commands (like **C-x n n** for narrowing) are disabled by default because they might be confusing for new Emacs users. Its possible to enable or disable commands using the following commands.		
Enable a command		(**enable-command** COMMAND)	Allow COMMAND to be executed without special confirmation from now on. COMMAND must be a symbol. • This command alters the user's .emacs file so that this will apply to future sessions. • It adds a *(put 'COMMAND 'disabled t)* inside the emacs init file.
Disable a command		(**disable-command** COMMAND)	Require special confirmation to execute COMMAND from now on. COMMAND must be a symbol. • This command alters your init file so that this choice applies to future sessions. • It adds a (put 'COMMAND 'disabled nil) inside the emacs init file.

Operation	Keystroke	Function	Notes
Libraries	The commands below are used to find and load Emacs Lisp **libraries**		
Load a Lisp library from load-path	• **<f12> l L** • **<M-f12> l L** • **<f11> SPC l l L**	(**load-library** LIBRARY)	Load the Emacs Lisp library named LIBRARY. • Emacs prompts for LIBRARY, a string, identifying the Emacs Lisp file: no need for the path or the extension, the file is searched searched for in '*load-path*', both with and without '*load-suffixes*' (as well as '*load-file-rep-suffixes*').
Find and open Library file	• **<f12> l l** • **<M-f12> l l** • **<f11> SPC l l l**	(**find-library** LIBRARY)	Find the Emacs Lisp source of LIBRARY. • Interactively, prompt for LIBRARY using the one at or near point.
Locate a library	• **<f12> l c** • **<M-f12> l c** • **<f11> SPC l l c**	(**locate-library** LIBRARY &optional NOSUFFIX PATH INTERACTIVE-CALL)	Show the precise file name of Emacs library LIBRARY. • LIBRARY should be a relative file name of the library, a string. • It can omit the suffix (a.k.a. file-name extension) if NOSUFFIX is nil (which is the default, see below). • This command searches the directories in '*load-path*' like '**<f11> SPC l l L**' to find the file that '**<f11> SPC l l L RET LIBRARY RET**' would load. • Optional second arg NOSUFFIX non-nil means don't add suffixes 'load-suffixes' to the specified name LIBRARY.
List available Emacs Lisp packages	• **<f12> l p** • **<M-f12> l p** • **<f11> SPC l l p**	(**package-list-packages** &optional NO-FETCH)	Display a list of packages. • This first fetches the updated list of packages before displaying, unless a prefix argument NO-FETCH is specified. • The list is displayed in a buffer named '*Packages*', and includes the package's version, availability status, and a short description.
Code Analysis	The commands below are used to **analyze** the Emacs Lisp code.		
Check validity of parentheses (or quotes, braces, brackets) (See also: 𝔓ι Common Lisp)	• **<f12>)** • **<M-f12>)** • **<f12> a)** • **<M-f12> a)** • **<f11> SPC l a)**	(**check-parens**)	Check for unbalanced parentheses in the current buffer. • More accurately, check the narrowed part of the buffer for unbalanced expressions ("sexps") in general. This is done according to the current syntax table and will find unbalanced brackets or quotes as appropriate. (See Info node '(emacs)Parentheses'.) If imbalance is found, an error is signaled and point is left at the first unbalanced character.
ELint the code in current buffer	• **<f12> a b** • **<M-f12> a b** • **<f11> SPC l a b**	(**pel-lint-elisp-file**)	Run lint on Emacs Lisp file in current buffer. • This uses Elint. • This will open all Emacs Lisp files referred by the current file (via calls such as require calls) but also the files used by Emacs, to complete the lint analysis.
Analyze the style and documentation of code in current buffer	• **<f12> a d** • **<M-f12> a d** • **<f11> SPC l a d**	(**checkdoc**)	Interactively check the entire buffer for style errors. • The current status of the check will be displayed in a buffer which the users will view as each check is completed. • When errors are detected the analysis pauses and the user can enter recursive edit mode to correct the current style error and then resume the analysis by exiting the recursive edit with **C-M-c**.
ELint a specific Emacs Lisp file.	• **<f12> a f** • **<M-f12> a f** • **<f11> SPC l a f**	(**elint-file** FILE)	Lint the file FILE. • Emacs prompts for the file name.
ParInfer EDiff **Diff current code before/.after ParInfer modifications** (See also: ⅀ Diff & Merge)	• **<f12> a D** • **<M-f12> a D** • **<f11> SPC l a D**	(**parinfer-diff**)	Diff current code and the code after applying Indent Mode in Ediff. Use this to browse and apply the changes. 📦 Requires the parinfer package. PEL activates this when the **pel-use-parinfer** customize variable is set to **t**.
Macro Expansion	The macrostep package provides the macrostep-expand command that expands the macro code inside the code buffer (temporary turning the buffer in read-only mode). 📦 This requires the macrostep package. Under PEL, activated with ***pel-use-macrostep*** customize variable.		
Expand macro form code with macrostep	• **<f12> M-m** • **<M-f12> M-m** • **<f11> SPC l M-m**	(**macrostep-expand** &optional TOGGLE-SEPARATE-BUFFER)	Expand the macro form following point by one step. • Enters 'macrostep-mode' if it is not already active, making the buffer temporarily read-only. If macrostep-mode is active and the form following point is not a macro form, search forward in the buffer and expand the next macro form found, if any. • With a prefix argument, the expansion is displayed in a separate buffer instead of inline in the current buffer. Setting 'macrostep-expand-in-separate-buffer' to non-nil swaps these two behaviors.

macrostep-mode keys	Progressively expand macro forms with **e**, collapse them with **c**, and move back and forth with **n** and **p**. Use **q** or collapse all visible expansions to quit and return to normal editing.
	`key binding`
	`--- -------`
	`= macrostep-expand`
	`c macrostep-collapse`
	`e macrostep-expand`
	`n macrostep-next-macro`
	`p macrostep-prev-macro`
	`q macrostep-collapse-all`
	`u macrostep-collapse`
	`DEL macrostep-collapse`
	`C-c C-c macrostep-collapse-all`
	`C-M-i macrostep-prev-macro`		

Compiling	The commands below are used to **compile** the Emacs Lisp source code into byte code (.elc files) and navigate across the byte-compilation errors. When errors are detected, they are shown in a buffer. You can also click on the error links or type return on them to move point to the code error location.		
Byte-compile file in current buffer	• **<f12> c b** • **<M-f12> c b** • **<f11> SPC l c b**	(**pel-byte-compile-file-and-load**)	Byte compile and load the current elisp file.

Byte-compile complete directory of Emacs Lisp files	• **<f12> c d**
	• **<M-f12> c d**
	• **<f11> SPC l c d**
		(**byte-recompile-directory** DIRECTORY &optional ARG FORCE)	Recompile every '.el' file in DIRECTORY *that needs recompilation*.
			• This happens when a '.elc' file exists but is older than the '.el' file. Files in subdirectories of DIRECTORY are processed also.
			• It's possible to specify the first argument interactively (but not the second):
			  • If the '.elc' file does not exist, normally this function *does not* compile the corresponding '.el' file. However, if the prefix argument ARG is 0, that means do compile all those files. A nonzero ARG means ask the user, for each such '.el' file, whether to compile it. A nonzero ARG also means ask about each subdirectory before scanning it.
			  • If the third argument FORCE is non-nil, recompile every '.el' file that already has a '.elc' file.
			☝ If you upgrade or change version of Emacs you may want to byte recompile all files even if the .elc files exist and are newer than their corresponding .el file. In that case you must delete the .elc files first and then use the **C-u 0** prefix.

Operation	Keystroke	Function	Notes
Byte compile specified Emacs Lisp file	• **<f12> c f** • **<M-f12> c f** • **<f11> SPC l c f**	(**byte-compile-file** FILENAME &optional LOAD)	Compile a file of Lisp code named FILENAME into a file of byte code. • Emacs prompts for the filename. • The output file's name is generated by passing FILENAME to the function 'byte-compile-dest-file' (which see). • With prefix arg (noninteractively: 2nd arg), LOAD the file after compiling.
Move to next compile error	• **C-x `** • **M-g n** • **M-g M-n**	(**next-error** &optional ARG RESET)	A prefix ARG specifies how many error messages to move; • negative means move back to previous error messages. • Just **C-u** as a prefix means reparse the error message buffer and start at the first error. ⚠ This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must byte-compile the file first.
Move to previous compile error	• **M-g p** • **M-g M-p**	(**previous-error** &optional N)	Prefix arg N says how many error messages to move backwards (or forwards, if negative). ⚠ This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must byte-compile the file first.
Debugging Emacs Lisp	Emacs comes with 2 debuggers: 1. debug : built in Emacs, always available, uses the *Backtrace* buffer to show backtrace of execution. 2. edebug: source level debugger, shows the execution right inside the source code buffer.		

Debug	There are several ways to debug using debug:
	• Instrument the code by placing a **(debug)** call acting as breakpoints into the code to inspect.
	  • Use the commands listed below to invoke or schedule the invocation of the debugger, or
	  • kill the Emacs process externally with: **pkill -SIGUSR2 -i emacs** which toggles debug-on-quit when Emacs is hung.
	🎛 Debugger customization user option variables that control the debugger behaviour:
	• **debug-on-error**:
	  • Non-nil means enter debugger if an error is signalled.
	  • Does not apply to errors handled by 'condition-case' or those matched by '**debug-ignored-errors**'.
	  • If the value is a list, an error only means to enter the debugger if one of its condition symbols appears in the list.
	  • When you evaluate an expression interactively, this variable is temporarily non-nil if '**eval-expression-debug-on-error**' is non-nil.
	  • The command '**toggle-debug-on-error**' toggles this.
	• **debug-on-next-call**:
	  • Non-nil means enter debugger before next 'eval', 'apply' or 'funcall'.
	• **debug-on-quit**:
	  • Non-nil means enter debugger if quit is signaled (**C-g**, for example).
	  • Does not apply if quit is handled by a 'condition-case'.
	• **inhibit-debugger**:
	  • Non-nil means never enter the debugger.
	  • Normally set while the debugger is already active, to avoid recursive invocations.		

Identify function to debug	• **<f12> d d** • **<M-f12> d d** • **<f11> SPC l d d**	(**debug-on-entry** FUNCTION)	Request FUNCTION to invoke debugger each time it is called. • When called interactively, prompt for FUNCTION in the minibuffer. • This works by modifying the definition of FUNCTION. If you tell the debugger to continue, FUNCTION's execution proceeds. If FUNCTION is a normal function or a macro written in Lisp, you can also step through its execution. FUNCTION can also be a primitive that is not a special form, in which case stepping is not possible. Break-on-entry for primitive functions only works when that function is called from Lisp. • Use **M-x cancel-debug-on-entry** to cancel the effect of this command. • Redefining FUNCTION also cancels it.
Cancel debugging of function	• **<f12> d D** • **<M-f12> d D** • **<f11> SPC l d D**	(**cancel-debug-on-entry** &optional FUNCTION)	Cancel the debugging of specified function: undo effect of **M-x debug-on-entry** on FUNCTION. • If FUNCTION is nil, cancel debug-on-entry for all functions. • When called interactively, prompt for FUNCTION in the minibuffer. • To specify a nil argument interactively, exit with an empty minibuffer.
Activate/disable debugger on error	• **<f12> d !** • **<M-f12> d !** • **<f11> SPC l d !**	(**toggle-debug-on-error** &optional INTERACTIVELY)	Toggle whether to enter Lisp debugger when an error is signaled. • In an interactive call, record this option as a candidate for saving by "Save Options" in Custom buffers.
Activate/disable debugger on quit	• **<f12> d)** • **<M-f12> d)** • **<f11> SPC l d)**	(**toggle-debug-on-quit** &optional INTERACTIVELY)	Toggle whether to enter Lisp debugger when **C-g** is pressed. • In an interactive call, record this option as a candidate for saving by "Save Options" in Custom buffers.
Debugger *Backtrace* buffer commands	When the debugger is invoked, a *Backtrace* buffer window opens which displays the Lisp stack. Each line represents a function call, the most recent at the top. With it it is possible to view pending Lisp expressions, check the value of variables and force functions to return specified values. The mode accepts the commands listed below. • Step through the debugger using **d** • Use **c** to skip over an evaluation • Use **e** to evaluate a variable of interest in the concept of the code, or: hit **RET** with the cursor over the variable to evaluate it • Sexp can be evaluating within the calling context. • Provide a sexp to evaluate to function debug, showing the value when the debugger is opened.		
Step through	**d**	(**debugger-step-through**)	Proceed, stepping through subexpressions of this expression. Enter another debugger on next entry to eval, apply or funcall.
Continue	**c**	(**debugger-continue**)	Continue code execution - leave the debugger. • This is not available when the debugger was invoked because of an error.
Jump	**j**	(**debugger-jump**)	Continue to exit from this frame, with all debug-on-entry suspended.
Show/Hide variable	**v**	(**debugger-toggle-locals**)	Show or hide local variables of the current stack frame.
Evaluate expression	**e**	(**debugger-eval-expression** EXP &optional NFRAME)	Eval an expression, in an environment like that outside the debugger. The environment used is the one when entering the activation frame at point.
Display and Record expression	**R**	(**debugger-record-expression** EXP)	Display a variable's value and record it in '*Backtrace-record*' buffer.
Return value	**r**	(**debugger-return-value** VAL)	Continue, specifying value to return. • This is only useful when the value returned from the debugger will be used, such as in a debug on exit from a frame.
Debug frame	**b**	(**debugger-frame**)	Request entry to debugger when this frame exits. • Applies to the frame whose line point is on in the backtrace. • **Break** when returning from current function, continuing execution for the body of the function.
Cancel Debug frame	**u**	(**debugger-frame-clear**)	Do not enter debugger when this frame exits. • Applies to the frame whose line point is on in the backtrace.
Quit	**q**	(**top-level**)	Quit the debugger. Abort pending operation. Close the window and return point to previous location.
List functions that have debug on entry	**d**	(**debugger-list-functions**)	Display a list of all the functions now set to debug on entry.

Operation	Keystroke	Function	Notes
EDebug	Emacs edebug is a source level debugger, used within the Emacs Lisp source code. It shows more than the stack frame, putting a cursor in the source code where the break point is located. ☛ Edebug can be used to step though the code or not stop at all and gather execution coverage and frequency data. ☛ Once EDebug stops at a breakpoint the key binding of the EDebug commands that can only be used within the buffer currently in edebug-mode (ie. where EDebug is active) are shown in coral color. Some of the commands can also be issued from other buffers with different key bindings (and those are show in black). ☛ When an Emacs Lisp buffer has entered edebug-mode its mode line shows *Debugging* right beside the major mode.		
Instrumenting for Edebug	To use EDebug, first instrument the function(s) you want the debugger to step into: • Put point within or just after the function definition and type one of **C-u C-M-x** or **≈**. • It is also possible to instrument all definitions in a buffer and even all forms in a buffer. Options must be activated for that using (edebug-all-defs) or (edebug-all-forms). • To remove instrumentation from the function definition, simply re-evaluate the function definition with a command that does not instrument it, like **eval-last-sexp** with **C-x C-e**.		

Instrument most forms for Edebug (with variable controlling behaviour)	**C-u C-M-x**	(**eval-defun** EDEBUG-IT)
		
		– – – – – – – – – – – –
		
		(**edebug-eval-defun** EDEBUG-IT)
			Evaluate the top-level form containing point or after point and instrument for debugging if EDEBUG-IT is non-nil (which occurs when the **C-u** prefix argument is used).
			• The very first time (eval-defun t) is executed it loads edebug.el and advise eval-defun to edebug-eval-defun.
			• The following variables provide extra control:
			  • If **edebug-all-defs** is non-nil, that inverts the meaning of the prefix argument: in that case C-M-x instruments the definition unless it has a prefix argument. Its default is nil.
			  • If **edebug-all-defs** is non-nil, then the commands eval-region, eval-current-buffer and eval-buffer also instrument any definition they evaluate.
			  • If **edebug-all-forms** control whether eval-region should instrument any form, even non-defining forms. This does not apply to loading or evaluation in the minibuffer.

Toggle instrumenting for EDebugging of all definitions		(**edebug-all-defs**)	Toggle edebugging of all definitions that could be done by eval-region, eval-current-buffer and eval-buffer.
Toggle instrumenting for EDebugging of all forms		(**edebug-all-forms**)	Toggle edebugging of all forms.
Instrument top level form (always) for Edebug	• **<f12> d e** • **<M-f12> d e** • **<f11> SPC l d e**	(**edebug-defun**)	Evaluate the top level form point is in, stepping through with Edebug. • This is like 'eval-defun' except that it steps the code for Edebug before evaluating it. It displays the value in the echo area using 'eval-expression' (which see). • If you do this on a function definition such as a defun or defmacro, it defines the function and instruments its definition for Edebug, so it will do Edebug stepping when called later. It displays 'Edebug: FUNCTION' in the echo area to indicate that FUNCTION is now instrumented for Edebug. • If the current defun is actually a call to 'defvar' or 'defcustom', evaluating it this way resets the variable using its initial value expression even if the variable already has some other value. (Normally 'defvar' and 'defcustom' do not alter the value if there already is one.) • Instruments any top level form regardless of the value of edebug-all-defs and edebug-all-forms. edebug-defun is an alias for edebug-eval-top-level-form.
Instrument one more definition	**I**	(**edebug-instrument-callee**)	Instrument the definition of the function or macro about to be called (just after point). ☛ This command is only available when EDebug is active. • Do this when stopped before the form or it will be too late. • One side effect of using this command is that the next time the function or macro is called, Edebug will be called there as well. • If the callee is a generic function, Edebug will instrument all the methods, not just the one which is about to be called. Return the list of symbols which were instrumented.
EDebug Help	Once EDebug is active, use **?** to get help; a description of all available commands is listed on the Help buffer.		
Help	**?**	(**edebug-help**)	Describe 'edebug-mode'. Print the list of available Edebug commands inside a Help buffer.
Edebug Execution Modes	Once function(s) are instrumented, simply execute the code you want to debug. Once the debugger has reached a breakpoint Emacs enter the edebug-mode and the commands listed below are available. A quick overview, taken from the edebug.el source code state: • Step through the code with **SPC**, • Mark breakpoint with **b**, • Go until a breakpoint is reached with **g**, • Quit execution with **q**. • Use **?** to to describe other commands. The following commands correspond to EDebug execution modes (EDebug ways of operating — not related to the concept of Emacs minor/major modes). The commands in the list below run the program more slowly or stop sooner than the commands later in the list.		
Stop	**S**	(**edebug-stop**)	Stop execution and do not continue. • Useful for exiting from trace or continue loop.
Step	• **SPC** • **C-c C-s** • **C-x C-a C-s** • **C-x X SPC**	(**edebug-step-mode**)	Proceed to next stop point.
Next	• **n** • **C-c C-n** • **C-x C-a C-n**	(**edebug-next-mode**)	Proceed to next 'after' stop point.
Trace	• **t** • **C-x X t**	(**edebug-trace-mode**)	Begin trace mode: pause (normally 1 second) at each EDebug stop point. • Pauses for 'edebug-sit-for-seconds' at each stop point. • The trace can be interrupted by any key (like a navigation key or one of the EDebug command keys).
Trace Fast	• **T** • **C-x X T**	(**edebug-Trace-fast-mode**)	Trace with no wait at each step. • Updates the display at each stop point, but does not pause. • The trace can be interrupted by any key (like a navigation key or one of the EDebug command keys).
Go	• **g** • **C-x X g**	(**edebug-go-mode** ARG)	Go, evaluating until break: run until next breakpoint. • With prefix ARG, set temporary break at current point and go.
Continue	• **c** • **C-x X c**	(**edebug-continue-mode**)	Begin continue mode: pause one second at each breakpoint and then continue. • Pauses for 'edebug-sit-for-seconds' at each break point.
Continue Fast	• **C** • **C-x X C**	(**edebug-Continue-fast-mode**)	Trace with no wait at each step. • Updates the display at each break point, but does not pause.
Go Nonstop	• **G** • **C-x X G**	(**edebug-Go-nonstop-mode**)	Go, evaluating without debugging (ignoring the breakpoints). • You can also use 'edebug-stop', or any editing command, to stop.

Operation	Keystroke	Function	Notes
Controlling EDebug Execution Mode	By default EDebug stops at the first instrumented function it encounters. It can also be configured to stop only at the first breakpoint or never (useful for gathering coverage data). This is controlled by the value of the *edebug-initial-mode*. The possible values are: • step (the default) • go • Go-nonstop • some other EDebug options The following function can be used to change this.		
Change initial execution mode.	• `C-x C-a RET` • `C-x C-a C-m`	(**edebug-set-initial-mode**)	Set the initial execution mode of Edebug. • The mode is requested via the key that would be used to set the mode in edebug-mode. • This command prompts for the execution mode key, one of the single letters commands listed in the section above: SPC, n, t, T, g, c, C or G.
Edebug Jumping	The following commands execute until execution reach the specified location (or reach another breakpoint before). Except for step in they all create a temporary breakpoint for the intended destination. The commands, can, however, fail in case of nonlocal exit, bypassing reaching the temporary breakpoint. • The **f**, **o** and **h** commands display "Break" and pause for *edebug-sit-for-seconds* before showing the result of the form just evaluated. Setting this variable to nil suppresses this delay.		
Jump forward sexp	`f`	(**edebug-forward-sexp** ARG)	Proceed from the current point to the end of the ARGth sexp ahead. • If there is no Arg, jump forward 1 sexp • If there are not ARG sexps ahead, then do 'edebug-step-out'. ☛ If point is not located where the next step is, you can type **w** to move point there, before typing **f**. ⚠ Note that you must ensure that execution will go to the specified number of sexp, as it may not be the case if there are any conditional forms in the path.
Jump: step in	`i`	(**edebug-step-in**)	Step into the definition of the function, macro or method about to be called. • This first does 'edebug-instrument-callee' to ensure that it is instrumented. Then it does 'edebug-on-entry' and switches to 'go' mode. ☛ Once you step in a function with **i** it remains instrumented and will cause a stop upon future execution within the same Edebug session. To prevent this, simply re-evaluate the definition of that function to deinstrument it.
Jump: step out	`o`	(**edebug-step-out**)	Proceed from the current point to the end of the containing sexp. • If there is no containing sexp that is not the top level defun, go to the end of the last sexp, or if that is the same point, then step. • If the containing sexp is a function definition, this command continues until just before the last sexp in the definition. If it is already there, it returns from the function then stops. Essentially this command does not exit the currently executing function unless point is already positioned after its last sexp.
Goto here	`h`	(**edebug-goto-here**)	Proceed to first stop-point at or after current position of point. • Use this to execute up until a specific point (such as inside a specific condition) to see if execution gets there or when running a loop to see a specific value. • This does not set any breakpoint, so if you want to run again up to this location you can type **h** again on the same location.
EDebug Breakpoints	Edebug stops execution: 1. when the next stop point is reached (a stop point are before and after each form inside an instrumented function), 2. it reaches a breakpoint (which can be set and unset with the following first 3 commands) 3. on a **global break condition**, a conditional expression stored inside the edebug-global-break-expression (using the **X** command below) 4. on an **explicit source breakpoint**: a `(edebug)` call inside the source code. Note that breakpoints are ignored in the Go-non-stop mode (started with the **G** command, described above.		
Set breakpoint	• `b` • `C-x SPC` • `C-x X b`	(**edebug-set-breakpoint** ARG)	Set the breakpoint of nearest sexp. • With prefix argument, make it a temporary breakpoint (it's turned off the first time it stops execution). • This can be done at any time when Edebug is active
Unset breakpoint	• `u` • `C-c C-d` • `C-x X u`	(**edebug-unset-breakpoint**)	Clear the breakpoint of nearest sexp.
Set conditional breakpoint	• `x` • `C-x X x`	(**edebug-set-conditional-breakpoint** ARG CONDITION)	Set a conditional breakpoint at nearest sexp. • Emacs prompts for a condition. • The condition is evaluated in the outside context. • With prefix argument, make it a temporary breakpoint (it's turned off the first time it stops execution).
Move point to next breakpoint in current definition	`B`	(**edebug-next-breakpoint**)	Move point to the next breakpoint, or first if none past point.

Set global break condition	• `X`
	• `C-x X X`
		(**edebug-set-global-break-condition** EXPRESSION)	Set 'edebug-global-break-condition' to EXPRESSION.
			• The expression is tested at every stop point:
			  • if the result is non-nil, then break. Errors are ignored.
			  • This slows down execution, so if not needed set it to nil (the default).

Edebug Views	The following EDebug commands can be used to view aspects of the Emacs buffer and windows status as they were before entry to EDebug. These are is is useful when the code being debugged controls windows and buffers.		
View where am I	• `w` • `C-c C-l` • `C-x C-a C-l` • `C-x X w`	(**edebug-where**)	Show the debug windows and where we stopped in the program. ☛ This command is also used in the context of the Edebug Evaluation List buffer (see below) with the same behaviour.
Bounce to current point	`p`	(**edebug-bounce-point** ARG)	Bounce the point in the outside current buffer. • If prefix argument ARG is supplied, sit for that many seconds before returning. The default is one second.
View outside window	• `P` • `v`	(**edebug-view-outside**)	Change to the outside window configuration. • Use 'edebug-where' to return.
Toggle save windows	• `W` • `C-x X W`	(**edebug-toggle-save-windows** ARG)	Toggle the saving and restoring of windows. • With prefix, toggle for just the selected window. • Otherwise, toggle for all windows.
Evaluation in Edebug	When Emacs is in Edebug mode you can use the following commands to evaluate expression within the "***outside context***", the context of the program being debugged, as opposed to the context of EDebug itself (with some limitations — see the link). For instance when you evaluate an expression, you would not want it to be affected by the operations you performed during EDebug mode (liek the commands you issued). So EDebug saves some and restores the environment of the "program under test" when you evaluate an expression with the following commands.		
Eval Expression	`e`	(**edebug-eval-expression** EXPR)	Evaluate an expression in the outside context. • If interactive, prompt for the expression. • Print result in minibuffer.
Eval Last S-exp	`C-x C-e`	(**edebug-eval-last-sexp**)	Evaluate sexp before point in the outside context. • Print value in minibuffer.
Evaluate Expression in mini-buffer	`M-:`	(**eval-expression** EXP &optional INSERT-VALUE NO-TRUNCATE CHAR-PRINT-LIMIT)	Read a single Emacs Lisp expression in the mini buffer, evaluate it, and print the value in the echo area. • During EDebug session, this is done in the outside context.

Operation	Keystroke	Function	Notes
EDebug Evaluation List Buffer — evaluation watcher	When in edebug-mode you can use the **E** command to open a *edebug* buffer window where you can evaluate expression interactively within the "***outside context***" with the C-j and C-x C-e command just as you can in the *scratch* buffer. The only difference is that these are are EDebug specialized commands and they use EDebug "***outside context***". When debugging you may want to *watch* the value of some variables or expressions. Write these expressions inside the *edebug* buffer, in groups of 3 lines using the following layout but by creating them by writing the expression in the first line, evaluating it with C-j and then completing it with **C-c C-u**. You can repeat the operation several times with different expressions. The *edebug* buffer should contain 1 or several groups of 3 lines: • line 1: the expression under scrutiny • line 2: its value (you may use **C-j** the first time around to get the value • line 3: a Lisp comment (you may want to insert it yourself if the value is several lines. No need to add dashes (**C-c C-u** will do it). Once this is setup, return to the "program under test" with C-c C-w and continue the debugging (or tracing). You can the watch the expression changing values as execution of the "program under test" unfolds!		
Visit Eval List buffer	**E**	(**edebug-visit-eval-list**)	Switch to the evaluation list buffer "*edebug*".
Evaluate expression before point & insert value	**C-j**	(**edebug-eval-print-last-sexp**)	Evaluate sexp before point in outside environment; insert value. • This prints the value into current buffer.
Evaluate expression before point and print value in mini buffer	**C-x C-e**	(**edebug-eval-last-sexp**)	Evaluate sexp before point in the outside environment. • Print value in minibuffer.
Update the value of a watch group	**C-c C-u**	(**edebug-update-eval-list**)	Replace the evaluation list with the sexps now in the eval buffer.
Delete a watch group	**C-c C-d**	(**edebug-delete-eval-item**)	Delete the item under point and redisplay.
Return to the debugger	**C-c C-w**	(**edebug-where**)	Return to the the debug windows, where we stopped in the program.
Edebug Trace Buffer	By default during debugging nothing is stored in the trace buffer. To log execution of the stop points during debugging in the *debug-trace* buffer, set the *debug-trace* variable to non-nil. You can also use edebug-trace function in your code to trace information during execution of code even if Edebug is not active.		
Explicit call to trace		(**edebug-trace** FMT &rest ARGS)	Convenience call to 'edebug-trace-display' using 'edebug-trace-buffer'. ☛ This is **not** an Emacs command; it's function you can use in your code to force an explicit trace log.
EDebug Coverage Testing Support	Edebug provides rudimentary coverage testing and display of execution frequency. Each form is considered covered if it has returned two different values since the beginning of testing. This must be enabled by setting the ***edebug-test-coverage*** variable to non-nil. At the end use the **C-x X =** to put coverage comments inside source code (use one undo to remove it all).		
Display Freq Count	**C-x X =**	(**edebug-display-freq-count**)	Display the frequency count data for each line of the current definition. • The frequency counts are inserted as comment lines after each line, and you can undo all insertions with one 'undo' command. • The counts are inserted starting under the '(' before an expression or the ')' after an expression, or on the last char of a symbol. The counts are only displayed when they differ from previous counts on the same line. • If coverage is being tested, whenever all known results of an expression are 'eq', the char '=' will be appended after the count for that expression. Note that this is always the case for an expression only evaluated once. • To clear the frequency count and coverage data for a definition, reinstrument it.
Other Edebug commands	The following commands are available stop EDebug or view results that were printed in the minibuffer.		
Abort	• **a** • **C-]** • **C-x X a**	(**abort-recursive-edit**)	Abort the command that requested this recursive edit or minibuffer input.
Quit to top level	• **q** • **C-x X q**	(**top-level**)	Exit all recursive editing levels. However, instrumented code protected with ***unwind-protect*** or ***condition-case*** forms may resume debugging. • This also exits all active minibuffers.
Quit Nonstop	• **Q** • **C-x X Q**	(**edebug-top-level-nonstop**)	Set mode to Go-nonstop, and exit to top-level: don't stop even for protected code. • This is useful for exiting even if 'unwind-protect' code may be executed.
Previous result	**r**	(**edebug-previous-result**)	Print the previous result.
Show Backtrace	**d**	(**edebug-backtrace**)	Display a backtrace that is just a list of function calls. This is not a complete backtrace like you get with the debug system. But, as documented it is "Better than nothing…"
Profiler	Emacs has a built-in profiler that can be started with the command below and a command to stop it and get a report. No instrumentation is required to use this standard profiler. **Workflow:** 1. Start profiler with: **M-x profiler-start** 2. Execute code that must be profiled 3. Open the report with: **M-x profiler-report** 4. Stop the profiler with: **M-x stop-profiler** 5. To reset all data before profiling again: **M-x profiler-reset**		
Start the profiler		(**profiler-start** MODE)	Start/restart profilers. • MODE can be one of 'cpu', 'mem', or 'cpu+mem'. • If MODE is 'cpu' or 'cpu+mem', time-based profiler will be started. • Also, if MODE is 'mem' or 'cpu+mem', then memory profiler will be started.
Open profiler report.		(**profiler-report**)	Report profiling results. The report is opened in a **XX*-Profiler-Report *Date Time** buffer where the XX corresponds to the mode selected when the profiler was started, and the Data and Time correspond to the date/time of the report. The report looks like a outline tree with values and percentage to help identify what consumes the most.
Stop the profiler		(**profiler-stop**)	Stop started profilers. Profiler logs will be kept.
Reset the profiler		(**profiler-reset**)	Reset profiler logs.
Open profile file		(**profiler-find-profile** FILENAME)	Open profile FILENAME.

Operation	Keystroke	Function	Notes

ELProfiler	A separate profiler was written by Barry Warsaw: elp. The ELP package provides several functions to instrument code for profiling. This profiler is much more flexible but code must be instrumented and you must identify what functions to profile (with the elp-instrument- functions). You can also identify a "master" function: the profiler will only capture data during the execution of that function. There can be only one master function.
	To use the profiler, select the functions to instrument by using one of the tree **`elp-instrument-`** functions. This profiler allows you to concentrate on specific functions and ignore the remainder of Emacs.
	🎛 ELProfiler customization user option variables:
	• **elp-reset-after-results**: controls whether information is reset after display:
	  • Non-nil means reset all profiling info after results are displayed.
	  • Results are displayed with the 'elp-results' command.
	• **elp-use-standard-output**: control profiler output:
	  • If non-nil, output to 'standard-output' instead of a buffer.
	• **elp-sort-by-function**: control report ordering:
	  • Non-nil specifies ELP results sorting function. These functions are currently available:
	  • 'elp-sort-by-call-count' -- sort by the highest call count
	  • 'elp-sort-by-total-time' -- sort by the highest total time
	  • 'elp-sort-by-average-time' -- sort by the highest average times
	  • You can write your own sort function. It should adhere to the interface specified by the PREDICATE argument for 'sort'. Each "element of LIST" is really a 4-element vector where:
	  • element 0 is the call count,
	  • element 1 is the total time spent in the function,
	  • element 2 is the average time spent in the function,
	  • and element 3 is the symbol's name string.		

Instrument all functions in a package		(**elp-instrument-package** PREFIX)	Instrument for profiling, all functions which start with PREFIX. • For example, to instrument all ELP functions, do the following: `M-x elp-instrument-package RET elp- RET`
Instrument a function		(**elp-instrument-function** FUNSYM)	Instrument FUNSYM for profiling. • FUNSYM must be a symbol of a defined function.
Instrument a set of functions provided in a list		(**elp-instrument-list** &optional LIST)	Instrument, for profiling, all functions in 'elp-function-list'. • Use optional LIST if provided instead. • If called interactively, prompt for LIST in the minibuffer; type "nil" to use 'elp-function-list'.
Set the profile master function		(**elp-set-master** FUNSYM)	Set the master function for profiling. • This is not required, but if done it forces the profiler to only gather profiling data for the functions called during the execution of that master function. Useful when there's a need to profile the execution of a given function tree under a specific condition.
Stop using a master function		(**elp-unset-master**)	Unset the master function.
Remove the instrumentation in all instrumented functions		(**elp-restore-all**)	Restore the original definitions of all functions being profiled.
Remove instrumentation in a function		(**elp-restore-function** FUNSYM)	Restore an instrumented function to its original definition. • Argument FUNSYM is the symbol of a defined function.
Remove instrumentation in a set of functions provided in a list		(**elp-restore-list** &optional LIST)	Restore the original definitions for all functions in 'elp-function-list'. • Use optional LIST if provided instead.
After profiling, display the results		(**elp-results**)	Display current profiling results. • If 'elp-reset-after-results' is non-nil, then current profiling information for all instrumented functions is reset after results are displayed.
Reset profiling information for all instrumented functions		(**elp-reset-all**)	Reset the profiling information for all functions being profiled.
Reset profiling information for specific function		(**elp-reset-function** FUNSYM)	Reset the profiling information for FUNSYM.
Reset profiling information for the list of specified functions		(**elp-reset-list** &optional LIST)	Reset the profiling information for all functions in 'elp-function-list'. • Use optional LIST if provided instead.
ESUP - Emacs Start Up Profiler	The ESUP package is a specialized profiler: it profiles Emacs startup only: code called from the init.el file. Very useful to find what is slowing down Emacs on startup. ESUP profiles Emacs startup time by launching a new Emacs process from Emacs and examining all code executed at startup. 📦 This package is not part of Emacs distribution, to use it it must be installed. 🔄 To activate it under PEL the **pel-use-esup** customization variable must be set to **t**. To use: • Run on graphics mode only. The 'esup' command does not support Emacs running in terminal mode. • Type: **`M-x esup`** • Wait for an *esup* buffer to open with the results.		
Profile Emacs startup code	`M-x esup`	(**esup** &optional INIT-FILE &rest ARGS)	Profile the startup time of Emacs in the background. • If INIT-FILE is non-nil, profile that instead of USER-INIT-FILE. • ARGS is a list of extra command line arguments to pass to Emacs. ☛ 'esup' does not work in terminal mode. It must be executed in graphics mode. ⚠ 'esup' steps into `require' and `load' forms at the top level of a file but not if they are enclosed in any other statements. This limits its usefulness when conditional loading is located in the init.el file and when the use-package macros are used as well.
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside Emacs Lisp source code comments. This can be useful when using these markup languages to describe UML diagrams or finite-state machines for example.		

Preview UML diagram from plantUML source in current plantUML region of commented source code
 	
(See also: MJPlantUML)
	`<f12> u`	(**pel-render-commented-plantuml** PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment.
			• Use region if identified otherwise use PlantUML block at point.
			• Uses prefix (as PREFIX) to choose where to display it:
			  • 4 (when prefixing the command with **`C-u`**) -> new window
			  • 16 (when prefixing the command with **`C-u C-u`**) -> new frame.
			  • else -> new buffer
			• This can be used inside buffer using **any** major mode, when PlantUML markup is embedded inside source code comment.
			☝ Use this in source code to describe your code architecture with PlantUML markup, then generate the UML rendering by moving point inside the PlantUML block and issuing this command.
			📦 Requires the **plantuml-mode** external package, 🔄 activated by **pel-use-plantuml** user option being non-nil.

Operation	Keystroke	Function	Notes
Preview diagram created from Graphviz DOT markup embedded in comments (See also: ⅄Graphviz Dot)	`<f12> G`	(**pel-render-commented-graphviz-dot** &optional POS)	Render the Graphviz-Dot markup embedded in current mode comment. Search at POS if specified, otherwise search around point. Use region if identified otherwise use Graphviz-Dot block. ☝ The graphviz DOT code must be located within a block delimited by the following special keywords (that are also in comments): • **@start-gdot** • **@end-gdot** ⚠ The current implementation leaves the created image file in a temporary directory. You will probably want to move that file or delete it, otherwise the size of this directory will increase with each of these created files. The file names use the pel-gdot- prefix. 📦 Requires the **graphviz-dot-mode package** external package, 🛂 activated by **pel-use-graphviz-dot** user option set to **t**.

Emacs Lisp — Reference

Topic & link	Description
Books	
Writing GNU Emacs Extensions - O'Reilly by Bob Glickstein, July 2010	A good book that provides insight on how to use the various facilities to write good Emacs Lisp code. Emacs has evolved since the book was written but almost everything in the book still applies as of Emacs version 26.
Lisp Style	
Lisp Indentation Style @ Wikipedia	The Lisp Style is shown for some Common Lisp code but also applied to C and happens to be also very similar to the Python style (although in Python the blocks are simply indented; no parens character is used).
Lisp Editing - Parenthesis Highlighting	Several Emacs packages have been written to help highlight the parens. Emacs packages and modes include show-paren-mode, rainbow-delimiters and paren-face. PEL uses show-paren-mode and rainbow-delimiters
show-paren mode @ Emacs Manual	The paren.el is part of Emacs and implements the show-paren mode, which highlights the parens that matches the one before or after point.
rainbow-delimiters @ GitHub	The rainbow-delimiters mode allows colouring rareness according to their depth. When Emacs is used in Graphics mode it's also possible to assign different sizes as shown by Xah Lee in the ErgoEmacs Colored Nested Brackets page. The EmacsWiki Rainbow Delimiters page describes how to setup hooks that activate the mode automatically for some files.
paren-face @ GitHub	Defines a face named parenthesis used for the parentheses character, with the intention of dimming the parentheses to help show the real structure of Lisp code via indentation. The parinfer mode does something similar (if dims the closing parentheses).
Lisp Editing - Parenthesis Management	Several Emacs packages have been written to help the editing process. These include the following listed packages: adjust-parens, lispy, paredit, paxedit, parinfer, smartparens and probably several others.
Lisp Editing @ WikEmacs	This WikEmacs page describes several of those packages with editing scenarios
ParInfer	The parinfer package provides modes that infer the parenthesis.
ParInfer Documentation	The documentation allows live interaction
ParInfer Mode Implementation for Emacs (in Emacs Lisp)	Emacs Lisp code for ParInfer for Emacs. Describes how to install and configure ParInfer.
Highlighting Emacs Lisp Code	The default emacs-lisp-mode highlights the Emacs Lisp code available in the buffer. Emacs Lisp is a Lisp-2; so a symbol can be a variable and/or a function: each symbol has a link to variable definition, function definition and a property alist. Furthermore, there are different *kind* of functions: lambda, compiled-byte functions (autoloaded or not), macros (autoloaded or not), primitive (written in C), special forms (primitive written in C that treat the list differently). And there can be indirection and advices. There's also variation in the "kind" of variables: there's global variables, local variables, closures, etc… The standard highlighting does not show all of this information; the designers considered that it would be too distracting; just some of the information is available via highlighting. Some have different views and developed modes that highlight Emacs Lisp code differently. These modes are listed here.
highlight-defined @ MELPA	The highlight-defined package provides the highlight-defined-mode, a minor mode that highlights defined symbols. It has the ability to highlights differently different "*kind*" of function symbols. • Unfortunately it does not consider the semantic of the code enough in the selection of the highlighting. For example if you define a macro named while-n, the face you specify for macros won't be used for code that invokes the macro in a macro call form, however it will use that face if you specify a symbol like 'while-n in any list position except the first one. That mean it will be highlighted in the argument list (but not if the symbol is the first argument). • I would prefer highlighting to follow the code semantics, and perhaps have a customization option to colonize the arguments & variables that use the same name as functions. It might be difficult to do this in a minor mode. I'll have to investigate more.
The Emacs Lisp Mode Syntax Coloring Problem — Xah Lee	Xah Lee describes the problem he saw in the colouring. He tried to request changes to the Emacs developers, create a bug report and that was closed. So He wrote his own code. It's a new major mode, listed in the next row.
xah-elisp-mode @ MELPA	
Debugging Emacs Lisp	
An Introduction to Programming in Emacs Lisp - Debugging	A gentle introduction/overview of debugging Emacs Lisp with both debug and edebug, with examples.
GNU Emacs Lisp Manual: Debugging Lisp Programs	Extensive description of both debug and edebug.
How to debug elisp? @ stackOverflow	A discussion on debugging Emacs Lisp for a very quick oveview. Contribution from Drew Adams, Trey Jackson and Artur Malabarba.
Debugging Basics - Nic Ferrier's Youtube video	A 11 minute video showing a simple debugging session with edebug. Aside from the keyboard noise I find annoying, this video gives a good introduction of what can be done with EDebug, and also covers debugging of macros using macrostep to expand the macro before debugging to be able to see the execution inside the macro code.
Profiling Emacs Lisp	
GNU Emacs Lisp Manual: Profiling	Brief description of the built-in profiler and the elp package.
EmacsWiki - Emacs Native Profiler	List more functions than the GNU manual…
EmacsWiki - Emacs Lisp Profiler	Better description of the elp profiler.
Test Coverage	