

Programming Language Support — Common Lisp 🚧





























Description	Keystroke	Function	Note
Using lisp-mode to edit Common Lisp code	<div>🚧 Common Lisp mode is supported by a package called slime. The documentation of is table is not completed and much more needs to be tested.</div> <ul style="list-style-type: none">By default files with .l, .lsp or .lisp extensions are identified as general-purpose lisp files and open with the lisp-mode major mode.In this mode, use the C-M-x key with point at a define to send the define form to an exterior Lisp process identified by the inferior-lisp-program variable, and which is tied to the Emacs buffer identified by the inferior-lisp-buffer variable.Most people that write on this use the SBCL implementation of Common Lisp. I tried Clozure CL but the swank server did not run (🚧TODO: get CCL running with Slime). I have also tried the GNU CLISP implementation. This works but GNU CLISP only provides byte-compilation, not native compilation.PEL provides the following set of mode-specific key prefixes: <f11> SPC L , <f12> and <M-f12> The first one is always available. The other two prefixes are only available in lisp-mode buffers. The <M-f12> prefix helps the typing flow when the next key is a Meta key. For simplification, the <f11> SPC L prefix is normally omitted in the tab		
Open this PDF file. See also: 🔗 Help/Info	<div><f11> SPC L <f1></div> <div><f12> <f1></div>	<div>(pel-help-pdf &optional OPEN-WEB-PAGE)</div>	Open the local copy of the 📖 - Common Lisp PDF file unless a command prefix (like C-u) was used. In that case it opens the Github-hosted file instead.
🔗 Customize PEL Common Lisp support	<div><f11> SPC L <f2></div> <div><f12> <f2></div>	<div>(pel-customize-pel &optional OTHER-WINDOW)</div>	Customize PEL Lisp support: lisp, lispy. <ul style="list-style-type: none">If OTHER-WINDOW is non-nil (use C-u), display in another window.
🔗 Customize Emacs Common Lisp support	<div><f11> SPC L <f3></div> <div><f12> <f3></div>	<div>(pel-customize-library &optional OTHER-WINDOW)</div>	Customize Emacs Lisp support: lisp, lispy. <ul style="list-style-type: none">If OTHER-WINDOW is non-nil (use C-u), display in another window.
Using lisp-mode	M-x lisp-mode	(lisp-mode)	Major mode for editing Lisp code like Common Lisp. Automatically invoked for files with the following extensions: .l, .lsp or .lisp
Run Lisp Program	C-c C-z	(run-lisp CMD)	Run an inferior Lisp process, input and output via buffer “inferior-lisp”. <ul style="list-style-type: none">If there is a process already running in “inferior-lisp”, just switch to that buffer.This runs the exterior program identified by the variable <i>inferior-lisp-program</i>.<ul style="list-style-type: none">By default, the value of this program is: “lisp”.
🚧	C-c C-z	(switch-to-lisp EOB-P)	Switch to the inferior Lisp process buffer. With argument, positions cursor at end of buffer. Shadowed by another mode when in graphics mode running slime.
🚧	C-c C-z	(slime-switch-to-output-buffer)	Select the output buffer, when possible in an existing window. Hint: You can use ‘display-buffer-reuse-frames’ and ‘special-display-buffer-names’ to customize the frame in which the buffer should appear.
Extra Modes	The following commands can be used to activate or toggle useful modes for Emacs Lisp editing, specially for helping dealing with parenthesis: <ul style="list-style-type: none">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 Lispy mode See also: 📖 - Lispy	<div><f12> M-L</div> <div><f11> SPC L M-L</div>	<div>(pel-lispy-mode &optional ARG)</div>	Toggle lispy-mode on/off. Lispy is a minor mode for navigating and editing LISP dialects. 📦 Requires lispy external package. 📖 PEL downloads, installs and configure it when pel-use-lispy user option is set to t. Please read the information on lispy web site. 🖥️ pel-lispy-mode calls lisp-mode but also prepares hydra, loaded dynamically with PEL. 🚧 PEL support is very basic. More to come to add keys for terminal mode.
Toggle show-paren mode on/off See also: 🔗 Highlight	<div><f12> M-9</div> <div><M-f12> M-9</div> <div><f11> SPC L M-9</div> <div><f11> h (</div>	<div>(show-paren-mode &optional ARG)</div>	Toggle visualization of matching parens (Show Paren mode). <ul style="list-style-type: none">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	<div><f12> M-r</div> <div><M-f12> M-r</div> <div><f11> SPC L m R</div> <div><f11> h R</div>	<div>(rainbow-delimiters-mode &optional ARG)</div>	Highlight nested parentheses, brackets, and braces with different colours according to their depth. <ul style="list-style-type: none">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 user option is set to t.
Toggle ParInfer mode on/off	<div><f12> M-i</div> <div><M-f12> M-i</div> <div><f11> SPC L M-i</div>	<div>(parinfer-mode &optional ARG)</div>	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 user option is set to t.
Toggle between ParInfer Indent Mode and Paren Mode	<div><f12> M-I</div> <div><M-f12> M-I</div> <div><f11> SPC L M-I</div>	<div>(parinfer-toggle-mode)</div>	Switch ParInfer mode between Indent Mode and Paren Mode. 🔗📦 Requires the parinfer package. 🔗📖 PEL activates this when the pel-use-parinfer user option is set to t.
	<ul style="list-style-type: none">⚠️ 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: <ol style="list-style-type: none">ParInfer Indent Mode:<ul style="list-style-type: none">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.ParInfer Paren Mode:<ul style="list-style-type: none">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.		
Toggle between Lisp modes	<div><f12> M-l</div> <div><M-f12> M-l</div> <div><f11> SPC L M-l</div>	<div>(pel-toggle-lisp-modes)</div>	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	<div><f12> M-s</div> <div><M-f12> M-s</div> <div><f11> SPC L M-s</div>	<div>(semantic-mode &optional ARG)</div>	Toggle parser features (Semantic mode). <ul style="list-style-type: none">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.

Description	Keystroke	Function	Note
Evaluate/Compile Common Lisp code	When Slime is not is activated, use the following commands to evaluate forms in a buffer that contains Common Lisp code.		
Compile current define form	C-c C-c	(lisp-compile-defun &optional AND-GO)	Compile the current defun in the inferior Lisp process. <ul style="list-style-type: none"> DEFVAR forms reset the variables to the init values. Prefix argument means switch to the Lisp buffer afterwards. Shadowed by another mode when in graphics mode running slime.
Compile all Lisp code in current buffer	C-c C-k	(lisp-compile-file FILE-NAME)	Compile a Lisp file in the inferior Lisp process. Shadowed by another mode when in graphics mode running slime.
Load a Lisp file	C-c C-l	(slime-load-file FILENAME)	Load the Lisp file FILENAME. <ul style="list-style-type: none"> Use while point is in a source code buffer. Emacs prompt for the file name. Shadowed by another mode when in graphics mode running slime.
Eval form and go to next one	C-c C-n	(lisp-eval-form-and-next)	Send the previous sexp to the inferior Lisp process and move to the next one. <p>➡ This is also bound when slime is active.</p>
Eval paragraph	C-c C-p	(lisp-eval-paragraph &optional AND-GO)	Send the current paragraph to the inferior Lisp process. <ul style="list-style-type: none"> Prefix argument means switch to the Lisp buffer afterwards. Shadowed by another mode when in graphics mode running slime.
Eval region	C-c C-r	(lisp-eval-region START END &optional AND-GO)	Send the current region to the inferior Lisp process. <ul style="list-style-type: none"> Prefix argument means switch to the Lisp buffer afterwards. Shadowed by another mode when in graphics mode running slime.
Evaluate Common Lisp code with Slime	Once Slime is activated, use the following commands to evaluate forms in a buffer that contains Common Lisp code.		
Evaluate last expression	C-x C-e	(slime-eval-last-expression)	Evaluate the expression preceding point.
Evaluate Common Lisp defun form	C-M-x	(lisp-eval-defun &optional AND-GO)	Send the current defun to the inferior Lisp process. <ul style="list-style-type: none"> DEFVAR forms reset the variables to the init values. Prefix argument means switch to the Lisp buffer afterwards.
Evaluate last expression in Slime REPL	C-c C-j	(slime-eval-last-expression-in-repl PREFIX)	Evaluates last expression in the Slime REPL. <ul style="list-style-type: none"> Switches REPL to current package of the source buffer for the duration. If used with a prefix argument (C-u), doesn't switch back afterwards.
Compile all Lisp code in current buffer	C-c C-k	(slime-compile-and-load-file &optional POLICY)	Compile and load the buffer's file and highlight compiler notes. <ul style="list-style-type: none"> With (positive) prefix argument the file is compiled with maximal debug settings (C-u). With negative prefix argument it is compiled for speed (M--). If a numeric argument is passed set debug or speed settings to it depending on its sign. Each source location that is the subject of a compiler note is underlined and annotated with the relevant information. The commands 'slime-next-note' and 'slime-previous-note' can be used to navigate between compiler notes and to display their full details.
After compilation, go to next compilation note.	M-n	(slime-next-note)	Go to and describe the next compiler note in the buffer. Open a "slime-compilation" buffer to describe the current detected problem. Note: for reason unknown to me yet, the description of slime-next-note does not list its key bindings, but the key binding list it.
After compilation, go to previous compilation note.	M-p	(slime-previous-note)	Go to and describe the previous compiler note in the buffer. Open a "slime-compilation" buffer to describe the current detected problem. Note: for reason unknown to me yet, the description of slime-previous-note does not list its key bindings, but the key binding list it.
Introspection	I tried slime-who-calls while using Slime with GNU CLisp and it did not work. who-calls-who is not yet implemented in CLisp. Next step is to test with SBCL implementation, which seems to be the most popular implementation.🐛🐛🐛		
Show all specializations of class	<ul style="list-style-type: none"> C-c C-w a C-c C-w C-a 	(slime-who-specializes SYMBOL)	Show all known methods specialized on class SYMBOL.
Show all binders of global variable	<ul style="list-style-type: none"> C-c C-w b C-c C-w C-b 	(slime-who-binds SYMBOL)	Show all known binders of the global variable SYMBOL.
Find who calls	<ul style="list-style-type: none"> C-c C-w c C-c C-w C-c 	(slime-who-calls SYMBOL)	Show all known callers of the function SYMBOL.
Show expanders of macro	<ul style="list-style-type: none"> C-c C-w m C-c C-w RET 	(slime-who-macroexpands SYMBOL)	Show all known expanders of the macro SYMBOL.
Show referrers of global variable	<ul style="list-style-type: none"> C-c C-w r C-c C-w C-r 	(slime-who-references SYMBOL)	Show all known referrers of the global variable SYMBOL.
Show setters of global variable	<ul style="list-style-type: none"> C-c C-w s C-c C-w C-s 	(slime-who-sets SYMBOL)	Show all known setters of the global variable SYMBOL.
Show functions called by	<ul style="list-style-type: none"> C-c C-w w C-c C-w C-w 	(slime-calls-who SYMBOL)	Show all known functions called by the function SYMBOL.
List callers	C-c <	(slime-list-callers SYMBOL-NAME)	List the callers of SYMBOL-NAME in a xref window.
List callees	C-c >	(slime-list-callees SYMBOL-NAME)	List the callees of SYMBOL-NAME in a xref window.
	C-M-.	(slime-next-location)	Go to the next location, depending on context. When displaying XREF information, this goes to the next reference.
	C-M-,	(slime-previous-location)	Go to the previous location, depending on context. When displaying XREF information, this goes to the previous reference.
Static Analysis			
Expand Macro form	C-c RET	(slime-expand-1 &optional REPEATEDLY)	Display the macro expansion of the form starting at point. <ul style="list-style-type: none"> The form is expanded with CL:MACROEXPAND-1 or, if a prefix argument is given, with CL:MACROEXPAND. If the form denotes a compiler macro, SWANK/BACKEND:COMPILER-MACROEXPAND or SWANK/BACKEND:COMPILER-MACROEXPAND-1 are used instead. The expansion is written inside a "slime-macroexpansion" buffer. <ul style="list-style-type: none"> Inside the "slime-macro-expansion" buffer you can further expand with C-c RET and use (undo) to close the expansion.
Disassemble	C-c M-d	(slime-disassemble-symbol SYMBOL-NAME)	Display the disassembly for SYMBOL-NAME. <ul style="list-style-type: none"> The disassembled code is shown inside the "slime-description" buffer. The output depends on the used Common Lisp backend: since GNU Clips is a byte compiler, only byte-code is shown. When a SBCL is used the assembly code is shown. If you use Common Lisp built-in statistical performance analyzer, the assembler code is annotated with performance notes from the analyzer.

Description	Keystroke	Function	Note
Inspect expression	C-c I	(slime-inspect STRING)	Eval an expression and inspect the result. <ul style="list-style-type: none"> Takes the expression at (or before) point, prompt to confirm it. On Return executes it and display result inside a "slime-inspector" buffer. <ul style="list-style-type: none"> It's often better to inspect the <i>symbol</i>, so it's best to put a single quote before the symbol at the prompt before hitting return. Inside the "slime inspector" buffer several keys are available to control the inspection. <ul style="list-style-type: none"> Use <f1> m to list all available commands and their key bindings. These include: RET : Inspect item at point 1 : pop-up inspection level
Debugging	The following commands help debug Common Lisp code. - These work under GNU CLisp		
Trace/unTrace	C-c C-t	(slime-toggle-fancy-trace &optional USING-CONTEXT-P)	Toggle trace for a specified function. Use function at point but prompt to confirm.
Using Slime to edit Common Lisp code			
	M-x slime	(slime &optional COMMAND CODING-SYSTEM)	Start an inferior^_superior Lisp and connect to its Swank server.
	M-x slime-mode		
Getting help	When editing a Common Lisp file in <code>lisp-mode</code> and <code>slime-mode</code> , with the <code>slime</code> back-end running, the following commands are available to get help. NOTE: for some reason I got the functions when running in graphic mode but not in terminal. I need to investigate more. 🐛		
Slime Mode Info	C-h i slime		Once slime is installed, the Slime manual is available in Info.
Lookup Common Lisp keywords	C-c C-d h	(slime-documentation-lookup)	Generalized documentation lookup. Defaults to <code>hyperspec</code> lookup: opens a topic page in the browser Emacs uses. Useful to search for Common Lisp topics, define, macros, etc. <ul style="list-style-type: none"> The URL used for lookup is identify inside the variable <i>common-lisp-hyperspec-root</i>. 🐛 With PEL, identify the location of the HyperSpec directory you want to use by writing it inside <code>pel-clisp-hyperspec-root</code> user option. By default the URL is set to the LispWorks HyperSpec documentation root page, but you can modify it to identify a local directory using a "file://" prefix like "file://~/docs/HyperSpec/". PEL expands the ~ special character and set the <i>common-lisp-hyperspec-root</i> variable.
Complete symbol at point	C-c <tab>	(completion-at-point)	Perform completion on the text around point. <ul style="list-style-type: none"> Search for available Common Lisp symbols. Includes the symbols defined in currently compiled and loaded code. Shows all possible competitions inside a "Completions" buffer. The completion method is determined by 'completion-at-point-functions', which for my session was set at: (tags-completion-at-point-function)
Show argument list	C-c C-a	(lisp-show-arglist FN)	Show the argument list of the defun/macro at point. Implementation: send a query to the inferior Lisp for the arglist for function FN.
Show symbol documentation	C-c C-d	(lisp-describe-sym SYM)	Shadowed by another mode when in graphics mode running slime. Send a command to the inferior Lisp to describe symbol SYM. See variable 'lisp-describe-sym-command'.
	C-c C-e	(lisp-eval-defun &optional AND-GO)	Send the current defun to the inferior Lisp process. <ul style="list-style-type: none"> DEFVAR forms reset the variables to the init values. Prefix argument means switch to the Lisp buffer afterwards. Shadowed by another mode when in graphics mode running slime.
Show function documentation	C-c C-f	(lisp-show-function-documentation FN)	Show docstring of function at point. Prompts.
Show variable documentation	C-c C-v	(lisp-show-variable-documentation VAR)	Show documentation of variable at point. Prompts. Send a command to the inferior Lisp to give documentation for function FN. See variable 'lisp-var-doc-command'.
		(slime-close-all-parens-in-sexp &optional REGION)	Balance parentheses of open s-expressions at point. Point must be located after the unfinished s-expression. <ul style="list-style-type: none"> Insert enough right parentheses to balance unmatched left parentheses. Delete extra left parentheses. 🐛(I noticed that it does not always do that: need to find why, is this a bug?) Reformat trailing parentheses Lisp-stylishly. If REGION is true, operate on the region. Otherwise operate on the top-level sexp before point.
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 'Σ'). 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 .		
SemEd - Kill			
Kill next Lisp S-expression See also: <ul style="list-style-type: none"> Σ Cut & Paste (CLKB sl2.lisp) 	<ul style="list-style-type: none"> C-M-k <f11> -] 	(kill-sexp &optional ARG)	<ul style="list-style-type: none"> No argument: kill the next sexp (or the current from the point forward). With negative sign: kill the previous sexp (the sexp backward). <ul style="list-style-type: none"> 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: <ul style="list-style-type: none"> Σ Cut & Paste 	<ul style="list-style-type: none"> C-M-⌫ <f11> - [(backward-kill-sexp &optional ARG)	Kill the sexp (balanced expression) preceding point. <ul style="list-style-type: none"> 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: Σ 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.
SemEd - Mark			
mark function See also: Σ Marking	C-M-h	(mark-defun &optional ALLOW-EXTEND)	Put mark at end of this defun, point at beginning. <ul style="list-style-type: none"> 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.

Description	Keystroke	Function	Note
mark sexp and balanced expressions See also: <ul style="list-style-type: none"> 🔗 Marking (CLCB s1.lisp) 	<ul style="list-style-type: none"> Esc C-@ C-M-@ C-M-SPC <f11> . x 	(mark-sexp &optional ARG ALLOW-EXTEND)	Set mark ARG sexps (and balanced expressions) from point. <ul style="list-style-type: none"> 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	<ul style="list-style-type: none"> M-= <f11> . = 	(er/expand-region ARG)	Increase selected region by semantic units. <ul style="list-style-type: none"> 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. <p>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.</p> <p>➡ Once M-= is typed, you can quickly type the following single keys in sequence:</p> <ul style="list-style-type: none"> = to expand the region, - to contract the region, 0 to reset the operation. <p>If you wait too long, then you have to use M-= again to continue the expansion, otherwise the region is de-activated.</p> <p>Note that you can also use the following key chords to control the contraction of the selected text without having to worry about time:</p> <ul style="list-style-type: none"> M- M-= to contract the region M-0 M-= to reset the operation. <p>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.</p> <p>📦 This requires the expand-region package.</p> <p>➡ 🖱 Under PEL, activated with <i>pel-use-expand-region</i> user option.</p> <p>➡ 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.</p>
Navigation in LISP This current list below describe the specialized commands only. See the others inside 🔗 Navigation			
• By definitions/xref	Move to the definition of the defun, defmacro, variable, etc... at point. See 🔗 Xref for more information.		
Find definition of identifier at point See also: 🔗 Xref	M-.	(slime-edit-definition &optional NAME WHERE)	Lookup the definition of the name at point. <ul style="list-style-type: none"> If there's no name at point, or a prefix argument is given, then the function name is prompted.
Go back to where M-. was last issued	M-,	(slime-pop-find-definition-stack)	Pop the edit-definition stack and goto the location.
• To next/previous top-level forms	Move to beginning /end of S-expression forms. Jump over comments. Can be defun, defer, defconst, defmacros, free-from S-exp, etc... <p>The following 'beginning-of-defun' and 'end-of-defun' are standard Emacs commands. They have limitations:</p> <ul style="list-style-type: none"> They only navigate across any top-level form. <ul style="list-style-type: none"> They do not discriminate between a defun, a defmacro or even an unless form or any other top-level form. They do not skip doc-strings unless you set open-paren-in-column-0-is-defun-start user option to ignore '(' in strings. PEL provides an additional commands, complementing the standard Emacs commands: <ul style="list-style-type: none"> pel-beginning-of-next-defun which moves forward to the beginning of the next form pel-end-of-previous-defun which moves backward to the end of the previous top-level form 		
Change defun navigation functions (toggle between Emacs default and PEL's)	<ul style="list-style-type: none"> <f12> M-N <M-f12> M-N <f11> SPC L M-N	(pel-toggle-paren-in-column-0-is-defun-start)	Toggle interpretation of a paren in column 0 and display new behaviour. <ul style="list-style-type: none"> It toggles the standard Emacs 'open-paren-in-column-0-is-defun-start' user option, between: <ul style="list-style-type: none"> Interpret '(' in column 0 as always stating a defun (even in strings) - the default. Ignore '(' in strings. A '(' in column 0 is not automatically interpreted as starting a defun.
Backward to beginning of defun See also: 🔗 Navigation	<ul style="list-style-type: none"> C-M-a C-M-<home> <f6> p <f6> <up> 	(beginning-of-defun &optional ARG)	Move backward to the beginning of a defun. <ul style="list-style-type: none"> With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. <p>➡ 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> handles Shift-marking fine in terminal mode.</p>
	<p>👉 By default Emacs treats all opening parenthesis character in the first column as a defun.</p> <ul style="list-style-type: none"> This causes this function to stop at function definition inside strings. The behaviour can be changed by setting the open-paren-in-column-0-is-defun-start user option to nil. <ul style="list-style-type: none"> PEL provides pel-toggle-paren-in-column-0-is-defun-start to toggle that user option. You can also change it dynamically with <f12> M-N. <p>⚠ Moves to beginning of next function of the same nesting level of the current location. It skips the functions and methods that are more deeply nested.</p>		
Forward to end of defun See also: 🔗 Navigation	<ul style="list-style-type: none"> <f12> <right> <M-f12> <right> <ul style="list-style-type: none"> C-M-e C-M-<end> <f6> <right> 	(end-of-defun &optional ARG)	Move forward to next end of defun. <p>With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun.</p> <p>➡ Shift marking is available in graphics mode, not in terminal mode (both keys). However <f6> <right> and <f12> <right> handle Shift-marking fine in terminal mode.</p> <p>⚠ This command moves to the end of the next top-level function or class.</p>
Forward to start of next defun	<ul style="list-style-type: none"> <f6> n <f6> <down> 	(pel-beginning-of-next-defun ARG)	Move forward to the beginning of the next top-level form: function definition, macros, etc.. <ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> Move back to previous position with M-`. <p>➡ Shift marking is available with <f6> <down></p>
	<p>👉 This command is generic and for Emacs Lisp, moves to the beginning of the next top-level form.</p> <ul style="list-style-type: none"> Complements what end-of-defun does. Moves forward to the beginning of the function definition, which is often what users of other editors expect. <p>👉 By default Emacs treats all opening parenthesis character in the first column as a defun.</p> <ul style="list-style-type: none"> This causes this function to stop at function definition inside strings. The behaviour can be changed by setting the open-paren-in-column-0-is-defun-start user option to nil. <ul style="list-style-type: none"> PEL provides pel-toggle-paren-in-column-0-is-defun-start to toggle that user option. You can also change it dynamically with <f12> M-N. 		
Backward to end of previous defun	<ul style="list-style-type: none"> <f12> <left> <M-f12> <left> <f6> <left>	(pel-end-of-previous-defun &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous function definition. <ul style="list-style-type: none"> 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. <ul style="list-style-type: none"> Move back to previous position with M-`. <p>➡ Shift marking is available.</p>

Description	Keystroke	Function	Note
<ul style="list-style-type: none"> To next/previous selected top-level form or defun or ... <p>★★</p>	<p>Move to beginning /end of specified S-expression forms. Jump over comments and docstrings. Can be defun, defer, defconst, defmacros, free-from S-exp, groups of them, etc...</p> <p>👉 PEL provides the following powerful commands: pel-elisp-beginning-of-next-form and pel-elisp-beginning-of-previous-forms.</p> <ul style="list-style-type: none"> Their behaviour depends on the value of the pel-elisp-target-forms, pel-elisp-user-specified-targets and pel-elisp-user-specified-targets2 user-options, as well as their corresponding global or buffer-local values if they exist. The user options give you the ability to select the type of targets. You can either select the standard behaviour (target the top level forms), or use one of the other 6 types of targets. These include moving to top-level defun form, to any defun form, to defun, defmacro, defsubst, defalias, defadvice forms, to include the eieio forms, the variable definition forms or specify you own set of forms (and those can include the require and provide forms). More information is available in the docstring of these user options. When your buffer is using the Common-Lisp major mode, use the <f12> <f2> key sequence to open the relevant customization buffer which will allow you to see and change the persistent or current session settings. <p>👉 PEL also provides specialized versions of these commands:</p> <ul style="list-style-type: none"> pel-elisp-beginning-of-next-defun which moves to the beginning of next defun, pel-elisp-beginning-of-previous-defun to the previous defun. pel-elisp-to-name-of-next-defun which moves to the <i>name</i> of the next defun, pel-elisp-to-name-of-previous-defun to the previous one. pel-elisp-to-name-of-next-form which moves to the <i>name</i> of the next form, pel-elisp-to-name-of-previous-form to the previous one. 		
<p>Change target form for commands:</p> <ul style="list-style-type: none"> <f12> <up> <f12> <down> <f12> <C-up> <f12> <C-down> <p>★★</p>	<ul style="list-style-type: none"> <f12> M-n <M-f12> M-n 	(pel-elisp-set-navigate-target-form &optional GLOBALLY)	<p>Select form navigation behaviour.</p> <p>Select the behaviour of the following navigation functions:</p> <ul style="list-style-type: none"> 'pel-elisp-beginning-of-next-form' and 'pel-elisp-beginning-of-previous-form'.
	<f11> SPC L M-n	<ul style="list-style-type: none"> Modifies the value of 'pel-elisp-target-forms' user-option only for the current buffer unless the GLOBALLY argument is non-nil, in which case it modifies the behaviour for all buffers. The change in behaviour does not persist across Emacs sessions. For persistent change, open the customization buffer with <f12> <f2>, modify the value of the pel-elisp-target-forms, pel-elisp-user-specified-targets and pel-elisp-user-specified-targets2 user-options and save the customize buffer. 	
<p>Forward to start of next definition form</p> <p>★★</p> <p>Configurable target:</p> <ul style="list-style-type: none"> all top-level forms top-level defun all defun all defun, defsubst, defmacros, ... all variable definition forms: defvar, defconst, defcustom, defgroup, ... etc... 	<ul style="list-style-type: none"> <f12> <down> <M-f12> <down> 	(pel-elisp-beginning-of-next-form &optional N TARGET SILENT DONT-PUSH-MARK)	<p>Move point forward to the beginning of next N top-level form.</p> <ul style="list-style-type: none"> The search is controlled by the value of 'pel-elisp-target-forms' pel-elisp-user-specified-targets and pel-elisp-user-specified-targets2 user options. That value can be changed for the current session, for all buffers or only for the current buffer by the command 'pel-elisp-set-navigate-target-form', bound to <f12> M-n. It can also be specified by the TARGET argument: specify one of the symbols valid for 'pel-elisp-target-forms'.
	<f11> SPC L <down>		<ul style="list-style-type: none"> The function skips over forms inside docstrings. If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success. On success, push original position on the mark ring unless DONT-PUSH-MARK is non-nil. Move back to previous position with M-`. ➡ Shift marking is available with <f12> <down> 👉 This command is the most flexible and can be configured to move like the next 2 commands. <ul style="list-style-type: none"> It moves forward but to the beginning of the function definition, which is often what users of other editors expect. 👉 By default Emacs treats all opening parenthesis character in the first column as a defun: these are top-level forms. <ul style="list-style-type: none"> You can change the behaviour: for example, to move to next define or any group of top-level or indented definition forms like defsubst, defmacro, defvar, etc.. The behaviour is customizable (use <f12> <f2> then select the pel-sexp-form-navigation group to access the relevant user-options: pel-elisp-target-forms', 'pel-elisp-user-specified-targets' and 'pel-elisp-user-specified-targets2'. The customization can be saved and then become persistent across Emacs sessions. You can also control the values of these 2 user-options for all buffers or for each buffer separately: <ul style="list-style-type: none"> You can change the values of these variables for a specific buffer or all buffers not yet configured by using the <f12> M-n command. It's possible to set up a buffer to use the <f12> <down> key sequence to move to the next defun only or any top-level form, or some other selection or s-expression forms. Or define your own selection in pel-elisp-user-specified-targets and 'pel-elisp-user-specified-targets2' user-options, then activate them only for a buffer with <f12> M-n 8 key sequence. 👉 To count & display # selected forms forward: use a large numeric argument to force a failure: the error message shows number of instances found. 👉 All of these commands push the point in the mark stack: use M-` to move back to where the point was before the command was issued.
Forward to the name of the next form definition	<ul style="list-style-type: none"> <f12> <C-down> <M-f12> <C-down> 	(pel-elisp-to-name-of-next-form &optional N)	<p>Move point to the name of next N defun form - at any level.</p> <ul style="list-style-type: none"> Skip over forms located inside docstrings. Leave point on the first character of the form name. Move back to previous position with M-`.
Forward to beginning of next defun form	<ul style="list-style-type: none"> <f12> <M-down> <f12> f n <M-f12> f n 	(pel-elisp-beginning-of-next-defun &optional N)	<p>Move point to the name of the next defun form, whether it is top-level or indented.</p> <ul style="list-style-type: none"> The function skips over forms inside docstrings. Move back to previous position with M-`. 🖱 This uses pel-elisp-beginning-of-next-form specifying 'defun-forms as target type. ➡ Shift marking is available with <f12> <M-down>
Forward to the name of the next defun definition	<ul style="list-style-type: none"> <f12> <C-M-down> <M-f12> <C-M-down> 	(pel-elisp-to-name-of-next-defun &optional N)	<p>Move point to the name of next N defun form - at any level.</p> <ul style="list-style-type: none"> Skip over forms located inside docstrings and other types of forms. Leave point on first character of defun name. Move back to previous position with M-`.
<p>Backward to start of previous definition form</p> <p>★★</p> <p>Configurable target:</p> <ul style="list-style-type: none"> all top-level forms top-level defun all defun all defun, defsubst, defmacros, ... all variable definition forms: defvar, defconst, defcustom, defgroup, ... etc... 	<ul style="list-style-type: none"> <f12> <up> <M-f12> <up> 	(pel-elisp-beginning-of-previous-form &optional N TARGET SILENT DONT-PUSH-MARK)	<p>Move point backward to the beginning of previous N top-level form.</p> <ul style="list-style-type: none"> The search is controlled by the value of 'pel-elisp-target-forms' user option. That value can be changed for the current session, for all buffers or only for the current buffer by the command 'pel-elisp-set-navigate-target-form', bound to <f12> M-n. It can also be specified by the TARGET argument: specify one of the symbols valid for 'pel-elisp-target-forms'. The function skips over forms inside docstrings. If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success. Move back to previous position with M-`. ➡ Shift marking is available <f12> <up>
	<f11> SPC L <up>		<ul style="list-style-type: none"> 👉 This command is the most flexible and can be configured to move like the next 2 commands. <ul style="list-style-type: none"> It moves backward but to the beginning of the function definition, which is often what users of other editors expect. 👉 By default Emacs treats all opening parenthesis character in the first column as a defun: these are top-level forms. <ul style="list-style-type: none"> You can change the behaviour: for example, to move to next define or any group of top-level or indented definition forms like defsubst, defmacro, defvar, etc.. The behaviour is customizable (use <f12> <f2> then select the pel-sexp-form-navigation group to access the relevant user-options: pel-elisp-target-forms', 'pel-elisp-user-specified-targets' and 'pel-elisp-user-specified-targets2'. The customization can be saved and then become persistent across Emacs sessions. You can also control the values of these 2 user-options for all buffers or for each buffer separately: <ul style="list-style-type: none"> You can change the values of these variables for a specific buffer or all buffers not yet configured by using the <f12> M-n command. It's possible to set up a buffer to use the <f12> <up> key sequence to move to the previous defun only or any top-level form, or some other selection or s-expression forms. Or define your own selection in pel-elisp-user-specified-targets and 'pel-elisp-user-specified-targets2' user-options, then activate them only for a buffer with <f12> M-n 8 key sequence. 👉 To count & display # selected forms backward: use a large numeric argument to force a failure: the error message shows # instances found.

Description	Keystroke	Function	Note
Backward to the name of the previous form definition	<ul style="list-style-type: none"> • <f12> <C-up> • <M-f12> <C-up> 	(pel-elisp-to-name-of-previous-form &optional N)	Move point to the name of previous N defun form - at any level. <ul style="list-style-type: none"> • Skip over forms located inside docstrings. Leave point on the first character of the form name. • Move back to previous position with M-`.
Backward to beginning of previous defun form	<ul style="list-style-type: none"> • <f12> <M-up> • <f12> f p • <M-f12> f p 	(pel-elisp-beginning-of-previous-defun &optional N)	Move point to the name of the previous defun form, whether it is top-level or indented. <ul style="list-style-type: none"> • The function skips over forms inside docstrings. • On success, push original position on the mark ring unless DONT-PUSH-MARK is non-nil. • Move back to previous position with M-`. •  Uses pel-elisp-beginning-of-previous-form specifying 'defun-forms as target type.  Shift marking is available with <f12> <M-up>
	<ul style="list-style-type: none"> • <f11> SPC L f p 		
Backward to the name of the previous defun definition	<ul style="list-style-type: none"> • <f12> <C-M-up> • <M-f12> <C-M-up> 	(pel-elisp-to-name-of-previous-defun &optional N)	Move point to the name of previous N defun form - at any level. <ul style="list-style-type: none"> • Skip over forms located inside docstrings and other types of forms. Leave point on first character of defun name. • Move back to previous position with M-`.
• By S-Expression form	Move across forms (S-expressions in Lisp).		
• By List element	• Move backward to the beginning or forward to the end of a S-expression form		
Backward block/list See also: ↗ Navigation	C-M-p	(backward-list &optional ARG)	Move backward across one balanced group of parentheses. <ul style="list-style-type: none"> • 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.
Move block backward See also: <ul style="list-style-type: none"> • ↗ Navigation • (CLCB s1.lisp) 	<ul style="list-style-type: none"> • C-M-b • C-M-<left> • C-[C-b • Esc C-b • Esc C-<left>  	(backward-sexp &optional ARG)	Move backward across one balanced expression (sexp). <ul style="list-style-type: none"> • 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. •  With PEL: if you want to use Esc C-<left> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil. •  C-M-<left> does not work on Windows, but H-<left> works.
	 With PEL: if you want to use Esc C-<left> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil.  Several Linux distros map C-M-<left> to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.		
Forward block/list See also: ↗ Navigation	C-M-n	(forward-list &optional ARG)	Move forward across one balanced group of parentheses. <ul style="list-style-type: none"> • 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.
Move block forward See also: <ul style="list-style-type: none"> • ↗ Navigation • (CLCB s1.lisp) 	<ul style="list-style-type: none"> • C-M-f • C-M-<right> • C-[C-f • Esc C-f • Esc C-<right>  	(forward-sexp &optional ARG)	Move forward across one balanced expression (sexp). <ul style="list-style-type: none"> • 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. •  With PEL: if you want to use Esc C-<right> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil. •  C-M-<right> does not work on Windows, but H-<right> does.
	 With PEL: if you want to use Esc C-<right> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil.  Several Linux distros map C-M-<right> to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.		
• in/out of lists	• Move in and out of list nested levels.		
Backward Up/inside sexp hierarchy See also: <ul style="list-style-type: none"> • ↗ Navigation • (CLCB s1.lisp) 	<ul style="list-style-type: none"> • C-M-u • C-M-<up> • C-[C-u • Esc C-u • Esc C-<up>  	(backward-up-list &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move backward out of one level of parentheses. <ul style="list-style-type: none"> • 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. •  With PEL: if you want to use Esc C-<up> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil. • 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.
Forward Up/outside sexp/block See also: ↗ Navigation	C-M-]	(up-list &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward out of one level of parentheses. <ul style="list-style-type: none"> • 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 to a less deep spot. • If ESCAPE-STRINGS is non-nil (as it is interactively), move out of enclosing strings as well. • If NO-SYNTAX-CROSSING is non-nil (as it is interactively), prefer to break out of any enclosing string instead of moving to the start of a list broken across multiple strings. On error, location of point is unspecified.
Forward Down/inside sexp/block See also: <ul style="list-style-type: none"> • ↗ Navigation • (CLCB s1.lisp) 	<ul style="list-style-type: none"> • C-M-d • C-M-<down> • C-[C-d • Esc C-d • Esc C-<down>  	(down-list &optional ARG)	Move forward down one level of parentheses. <ul style="list-style-type: none"> • 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. •  With PEL: if you want to use Esc C-<down> binding you must ensure that pel-windmove-on-esc-cursor user option is set to nil. • 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.
Search Support	In Common Lisp mode, the superword mode can be useful since snake_case is often used. Using superword-mode helps searching. PEL activates the superword mode by default in Common Lisp mode. To change this use the <f11> t <f2> to access the customize buffer.		
Toggle superword-mode See also: <ul style="list-style-type: none"> • ↗ Text Modes • ↗ Search/Replace 	<ul style="list-style-type: none"> • <f11> t m p • <f12> M-p 	(superword-mode &optional ARG)	Toggle superword-mode: a minor mode that treats snake_case as one word. In CommonLisp ‘_’ and ‘_’ are treated as part of words. <ul style="list-style-type: none"> • With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise. • PEL provides the <f12> M-p key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc...)

Description	Keystroke	Function	Note
SemEd - Transpose			
<p>Transpose two balanced expressions (sexps)</p> <p>See also: ☞ Transpose (CLCB s1.lisp)</p>	<ul style="list-style-type: none"> C-M-t <f11> t t x 	(transpose-sexps ARG)	<p>Transpose 2 balanced expressions (text enclosed in parenthesis, braces, square or angle brackets, quotes, back-quotes and double quotes) of the same of different types. Here they are globally identified as <i>sexpr</i>.</p> <ul style="list-style-type: none"> Unlike ‘transpose-words’, point must be between the two sexps and not in the middle of a sexp to be transposed. With non-zero prefix arg ARG, effect is to take the sexp before point and drag it forward past ARG other sexps (backward if ARG is negative). If ARG is zero, the sexps ending at or after point and at or after mark are interchanged.
SemEd - Indenting	<p>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 <i>lisp-indent-function</i> .</p> <p>For Common Lisp the function to use is <i>common-lisp-indent-function</i> . The slime-setup function adds the slime-lisp-mode-hook function to the lisp-mode-hook. The slme-lisp-mode runs the required following code to install the indenter for Common Lisp:</p> <pre>(set (make-local-variable lisp-indent-function) 'common-lisp-indent-function)</pre>		
Indent current line (or region)	<tab>	(indent-for-tab-command &optional ARG)	<p>Indent the current line or region, or insert a tab, as appropriate.</p> <ul style="list-style-type: none"> 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 (CLBC s3.lisp)	C-M-q	(indent-sexp &optional ENDPOS)	<p>Indent each line of the list starting just after point.</p> <ul style="list-style-type: none"> If optional arg ENDPOS is given, indent each line, stopping when ENDPOS is encountered.
		(prog-indent-sexp &optional DEFUN)	<p>Indent the expression after point.</p> <ul style="list-style-type: none"> When interactively called with prefix, indent the enclosing defun instead. <p>Shadowed by another mode when in graphics mode running slime.</p>
SemEd - Parentheses	The commands below are used to help dealing with the parentheses (along with the semantic editing navigation commands listed above).		
<p>Insert Parentheses</p> <p>(See also:</p> <ul style="list-style-type: none"> ☞I - Emacs Lisp CLCB s4.lisp 	M- ((insert-parentheses &optional ARG)	<p>Enclose following ARG sexps in parentheses.</p> <ul style="list-style-type: none"> 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.
<p>Move past close ‘)’ and reindent</p> <p>See also ☞I - Emacs Lisp</p>	M-)	(move-past-close-and-reindent)	<p>Move past next ‘)’, delete indentation before it, then indent after it.</p> <ul style="list-style-type: none"> Used to add another entry in the parent list.
<p>Check validity of parentheses</p> <p>(or quotes, braces, brackets)</p> <p>See also ☞I - Emacs Lisp</p>	<ul style="list-style-type: none"> <f12>) <M-f12>) 	(check-parens)	<p>Check for unbalanced parentheses (or quotes, braces and brackets) in the current buffer.</p> <ul style="list-style-type: none"> 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.
Close all parentheses of open expression at point	C-c C-]	(slime-close-all-parens-in-sexp &optional REGION)	<p>Balance parentheses of open s-expressions at point.</p> <ul style="list-style-type: none"> Insert enough right parentheses to balance unmatched left parentheses. Delete extra left parentheses. Reformat trailing parentheses Lisp-stylishly. If REGION is true, operate on the region. Otherwise operate on the top-level sexp before point. <p>TODO: check where this function is defined.🐛</p>
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside CommonLisp source code comments. This can be useful when using these markup languages to describe UML diagrams or finite-state machines for example.		
<p>Preview UML diagram from plantUML source in current plantUML region of commented source code</p> <p>See also: ☞ PlantUML</p>	<f12> u	(pel-render-commented-plantuml PREFIX &optional POS)	<p>Render the PlantUML markup embedded in current mode comment.</p> <ul style="list-style-type: none"> Use region if identified otherwise use PlantUML block at point. Uses prefix (as PREFIX) to choose where to display it: <ul style="list-style-type: none"> 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. <p>👉 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.</p> <p>📦 Requires the plantuml-mode external package, 🔗 activated by pel-use-plantuml user option being non-nil.</p>
<p>Preview diagram created from Graphviz DOT markup embedded in comments</p> <p>See also: ☞ Graphviz Dot</p>	<f12> G	(pel-render-commented-graphviz-dot &optional POS)	<p>Render the Graphviz-Dot markup embedded in current mode comment.</p> <p>Search at POS if specified, otherwise search around point.</p> <p>Use region if identified otherwise use Graphviz-Dot block.</p> <p>👉 The graphviz DOT code must be located within a block delimited by the following special keywords (that are also in comments):</p> <ul style="list-style-type: none"> @start-gdot @end-gdot <p>⚠️ 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.</p> <p>📦 Requires the graphviz-dot-mode package external package, 🔗 activated by pel-use-graphviz-dot user option set to t.</p>

Description & URL	Notes
Wikipedia — Lisp	The page for Lisp language family. List the Lisp family of languages, the main Lisp concepts and facilities.
Paul Graham — The way Lisp began	Describes the way John McCarthy developed the concepts of Lisp in 1960 and forward.
Common Lisp — The language	The following links refer to Common Lisp itself.
Wikipedia — Common Lisp	An overview of Common Lisp with several links.
Common Lisp HyperSpec	A Common Lisp reference, with hyperlinks accessing information from various angles. It is not a tutorial, but rather a specification and reference, but very useful when looking for specific details. The (slime-documentation-lookup) opens the web page corresponding to the topic requested. It is possible to get a local copy of the HTML files and set Emacs to use the local copy. See the LispWorks copyright notice for more details.
Common Lisp — Implementations	There are several implementation of Common Lisp, some commercial other open source. The open source one most popular to use with Slime is SBCL. GNU CLisp does not implement everything required for introspection.
Derek Banas Youtube Lisp Tutorial	A Common Lisp tutorial using GNU CLisp (and not Emacs!) but it helps getting a quick overview of Common Lisp. <ul style="list-style-type: none"> Note that this tutorial goes over concepts very quickly and sometimes does not emphasizes the important aspects of the areas covered. So don't use this as the sole source for learning Common Lisp!
Common Lisp Development Environment	
Setting up Lisp Environment @ Common-Lisp.net	They recommend using Emacs with SLIME as text editor/IDE and ASDF + Quicklisp for project setup and libraries. The site states (copied from the web site) <ul style="list-style-type: none"> SLIME is an extension to the Emacs text editor that connects the editor to the running Lisp image (called *inferior-lisp*) and interacts with it. It provides lisp code evaluation, compilation, and macroexpansion, online documentation, code navigation, objects inspection, debugger, and much much more. ADSF is the Lisp version of Make. It is used to define projects (called systems), its dependencies, and load and compile the project. Quicklisp is a library manager for Common Lisp. Use it to download, install, and load any of over 1,500 libraries with a few simple commands.
The Common Lisp Cookbook — Using Emacs as a Lisp IDE (2013)	A web page that describes several Common Lisp packages that can be used within Emacs. <ul style="list-style-type: none"> It describes how to use the various Slime commands with code example. It also provides a Q&A on how to do several things within Emacs wrt Common Lisp, for example how to get the Hyperspec show up inside Emacs instead of in a browser.
The Common Lisp Cookbook - Using Emacs as a Lisp IDE	Same as above, an older version, but holds links to source code example that are not present above.
Paredit	Apparently Emacs paredit allows you to become very efficient in writing Lisp code, although it is difficult to learn at first. Parentheses are never placed manually.
SLIME	SLIME allows you to compile Common Lisp code directly from Emacs. <ul style="list-style-type: none"> It uses a backend Common Lisp compiler that must be installed separately and identified by the <i>inferior-lisp-program</i> variable, and which is tied to the Emacs buffer identified by the <i>inferior-lisp-buffer</i> variable. The installation can be done via the Emacs M-x package-list-packages command either from MELPA or MELPA Stable. Once installed you can read the manual via the Info with C-h i and then select the Slime node. Note that after installing slime, you may have to close the *info* buffer and re-open it to see the slime info node. To use it, execute M-x slime on a buffer that holds a Common Lips source code file: it launches and connects to the backed Common Lisp server and activates the slime-mode for the Common Lips file buffer which complements the standard lisp-mode major mode used to edit Common Lisp code.
SLIME: The Superior Lisp Interaction Mode for Emacs	SLIME has 2 sides: one written in Emacs Lisp that connects to a Common Lisp backend.
slime 2.24 @ MELPA Stable	As of January 2, 2020, slime 2.24 is hosted at MELA stable. This corresponds to the code as it was May 27 2019. A later version is available at MELPA as this is actively maintained. I did not see major issues to mandate using a non-stable version.
Slime 2.24 manual	The Slime manual 2.24 is available inside Emacs Info (C-h i) top level.
Slime 2.22 Manual (html)	The latest version of the Slime manual located on the common-lisp.net web site
SLIME @ Github	Although the versions above are OK, if you want to participate in the development of SLIME, use the code from its depot.
Emacs Manual - Running and External Lisp	Describes the mode used to edit Common Lisp (and other dialects) of general-purpose Lisp code, how to evaluate functions defined in Common-Lisp by using an exterior Common Lisp process identified and used by Emacs. For example if a buffer contains Common Lisp source code and is using the lisp-mode major mode, then typing C-M-x while point is over a define form sends that form to the exterior Lisp process, allowing it to be used there.
Youtube - Emacs with Slime. Really useful keyboard shortcuts	A quick and easy to follow example of using Slime with SBCL. Worth watching.
Other Slime packages	
slime-ac	Slime autocomplete. Automatically completes current symbol. Note that without that package you can use C-c <tab> to get a completion list.
Lisp in a box	An old, an unmaintained, package that combined Emacs, SLIME, ASDF and Quicklisp. At this point in time (Jan 2020), it seems that it's better to install them separately.
Common Lisp Books	The following pages contain links to several books on Common Lisp and related subjects: <ul style="list-style-type: none"> lisp-lang.org Common Lisp Books Wikipedia Common Lisp Publications
The Common Lisp Cookbook	This is a book under development in the style of O'Reillys programming cookbooks. The page also contains links to several other Common Lisp resources.
Practical Common Lisp - by Peter Siebel	A good book to start learning Common Lisp. On line. with source code downloadable on the book site. Note that SLIME has evolved since the book was written. I did not find an errata for the book (yet). For example several key bindings and Emacs slime commands seems to have been renamed/modified since the book was written.
ANSI Common Lisp - by Paul Graham, 1995	
On Lisp: Advanced Techniques for Common Lisp - by Paul Graham	
Common Lisp the Language, 2nd Edition - by Guy L. Steele	This book, published in 1984 (1st edition) and 1990 (second edition) had a large influence on the ANSI standard (published in 1994). The Wikipedia page for the Common Lisp the Language book provides overview description and several links.
Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp — Peter Norvig, 1992	This book uses Common Lisp for very interesting AI topics, showing how to write good Lisp software. The link point to a github site that contains the book material since Peter Norvig released his book in various electronic formats along with source code in markdown format. The copyright was reverted to Peter Norvig who released it under a MIT license.
The Art of the Metaobject Protocol — MIT Press	
Common Lisp References	
Common Lisp ANSI Standard — INCITS 226-1994 (R1999) (formerly ANSI X3.226-1994)	
Common Lisp HyperSpec	The Wikipedia page for the Common Lisp HyperSpec provides links to the main page as well as the set of page data.
Common Lisp Topics - Debugging	
malisper.me Category: Debugging Lisp	Blog on debugging Common Lisp with Emacs, Slime and SBCL, written in 2015. This is a series of 5 articles.