↖ Topic Index

Programming Language Support — Common Lisp 🚧

Description	Keystroke	Function	Note
Common Lisp ○ Help, lisp-mode • useful minor-modes ○ Getting help: sly, slime, lispy • CL Hyperspec, Autodoc • Lisp REPL ○ sly connection ○ Evaluate code • macro expansion ○ Introspection • Static analysis • Debugging • Sly stickers ○ Semantic Editing • SemEd-Mark ○ Navigation in LISP • by definition/xref ○ to next/previous top-level form or defun ○ by S-expression • list element ○ in/out list • Search support • SemEd Transpose • SemEd Indentation ○ SemEd Parentheses • Rendering markup in comments • Using Quicklisp • Common Lisp Reference Last updated on: 2026-02-04			PEL provides Common Lisp support when the **pel-use-common-lisp** user-option is on (set to **t**). See **Common Lisp implementations** • This user-option is part of the **pel-pkg-for-clisp** customization group. • Aside from the usual ways to access it PEL provides the **<f11> SPC L <f2>** key binding to open it, see below. ***File association*** for **lisp-mode** major mode already identified by Emacs: • **.l**, **.lsp**, and **.lisp** : Common Lisp source files • **.ml** • **.asd** : ASDF (Another System Definition Facility) package formal and and Lisp build/packaging system • With PEL, you can define more files or file extensions by adding regular expressions in the **pel-clisp-extra-files** user-option. ☝ Add more file associations by putting them into **pel-auto-mode-alist** user-option. • PEL activates Σ **Speedbar** support for the lisp files when **pel-use-speedbar** user-option is on (set to **t**). • 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 table cells. **PEL imenu symbol extraction**: See also: Σ **Menus**, Σ **File/Directory Variables** • PEL provides the ability to extend Emacs imenu symbol extraction for Common Lisp source code, including inside ASDF files. • The **pel-clisp-define-forms** user-option defines how to extract Common Lisp define forms (`define-class`, `define-mode`, etc…) as well as the ASDF `defsystem` form. You can add and modify the rules by customizing this user-option to conform to your Common Lisp coding environment. • Like all customized variables, you can also define it inside an Emacs .dir-locals.el file providing more flexibility. **Selection of the Common Lisp process**: When either of these Common Lisp Emacs IDE is in place, they use external Common Lisp process by the ***inferior-lisp-program*** variable. • The default value is "lisp". PEL provides the **pel-inferior-lisp-program** user-option you can use to customize that value. • You could also create a symlink to your Common Lisp program and call it "lisp", or create a shell script called lisp that executes your process, possibly selecting it from some criteria you may have, allowing the use of multiple implementations of Common Lisp very simply. The Emacs buffer tied to the inferior lisp process is identified by the ***inferior-lisp-buffer*** variable. **Common Lisp Code Style**: adjustments available in the Common Lisp code style sub-group: **pel-clisp-code-style** • **pel-clisp-fill-column** : column where line-wrapping occurs: maximum line length (defaults to 100). Change to any integer or nil to use Emacs default. PEL support the following **Common Lisp IDE minor modes** activated by identified user-options:	
	📦 **slime**	**pel-use-slime** : **t** **pel-clisp-ide : slime**	By default Slime activates only the slime-fancy "*contrib*". To activate more slime features add them to the **pel-use-slime** user-option: select value 2 insert one or more slime "*contrib*" symbols.
	📦 **sly**	**pel-use-sly** : **t** **pel-clisp-ide : sly**	Another Common Lisp IDE system.
	📦 **lispy** (or **this fork**)	**pel-use-lispy** ☝ when changing value, move the old one out of the ~/.emacs.d/elpa directory.	Very useful, powerful, elegant minor mode to edit Lisp languages. See 𝔓ℓ- **Lispy** for more info. Set **pel-use-lispy** the following values to install a specific implementation: • **t** : use the original **abo-abo/lispy** (which has not been updated for a while) • **use-enzuru-lispy,** to use the temporary **enzuru fork.**
	📦 **parinfer** 📦 **parinfer-rust-mode**	**pel-use-parinfer**	Supports the **ParInfer** editing mode. Installs the package selected by **pel-use-parinfer** value: • **t**: installs **parinfer ,** a fork of the original code. • **use-parinfer-rust-mode**: **parinfer-rust-mode**, which use a back-end written in **Rust**.
	📦 **rainbow-delimiters**	**pel-use-rainbow-delimiters**	Minor-mode that highlight nested parentheses, brackets, and braces with different colours according to their depth.
Open this PDF file. See also: Σ **Help/Info**	**<f11> SPC L <f1>** **<f12> <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the 𝔓ℓ **- Common Lisp** local PDF. If the prefix argument (like **C-u** or **M--**) is used, then it opens the remote GitHub hosted raw PDF instead. If the **pel-flip-help-pdf-arg** user-option is set it's the other way around.
Σ **Customize** PEL Common Lisp support	**<f11> SPC L <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL Lisp support: lisp, lispy. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window. ☛ Use **<f11> SPC L <f2>** if PEL Common Lisp support is off. Once it's activated you can use the **<f12> <f2>** keybinding from a buffer using the lisp-mode.
Σ **Customize** Emacs Common Lisp support	**<f11> SPC L <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs Lisp support: lisp, lispy, slime, sly. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Using the lisp-mode			The major mode used for Common Lisp buffers is the **lisp-mode**. You can also use the **slime** or the **sly** external packages to enhance the available features. • You can activate it manually with the command below but as described above it be automatically invoked for Common Lisp files.
Using lisp-mode	`M-x lisp-mode`	(**lisp-mode**)	Major mode for editing Lisp code like Common Lisp. Used by **.l**, **.lsp** or **.lisp** files.
Useful Minor Modes			Several minor modes can be used to activate various features when editing Common Lisp files. • With PEL, activate a minor mode for Common Lisp files by adding its function name to the **pel-lisp-activtates-minor-modes** user-option.
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.
Toggle Lispy mode See also: 𝔓ℓ- **Lispy**	**<f12> M-L** **<f11> SPC L M-L**	(**pel-lispy-mode** &optional ARG)	Toggle lispy-mode on/off. Lispy is a minor mode for navigating and editing LISP dialects. • **pel-lispy-mode** calls **lispy-mode**, if enabling: disables **overwrite-mode** and prepares **hydra**. 📦 Requires **lispy** external package. PEL downloads, installs and configure it when **pel-use-lispy** user option is set to **t**.
Toggle show-paren mode on/off See also: Σ **Highlight**	**<f12> h (** **<f11> 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: Σ **Highlight**	**<f12> h)** **<f11> h)**	(**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** user option 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 one of the parinfer packages. PEL activates via **pel-use-parinfer** user option.
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. 📦 Requires parinfer external package. PEL activates it when **pel-use-parinfer** is set to **t**.
			⚠ 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.

Description	Keystroke	Function	Note
Getting Help • See also: Σ Help/Info	When editing a Common Lisp file in lisp-mode using slime-mode, with the slime back-end running, the following commands are available to get help. Once the packages are installed, you can get their manual using the **C-h i** or the **<f1> i** key sequences and then type their name.		
• about slime : Slime Mode	**C-h i slime**	(**info** &optional FILE-OR-NODE BUFFER)	Open the Slime Info Manual inside Emacs. ☛ Available once slime is installed. • See also this Slime Reference PDF card
• about sly : SLY	**C-h i SLY**		Open the SLY Info Manual inside Emacs. ☛ Available once sly is installed.
• about the code 1. With slime 2. With sly See also: Ψ- Lispy	Use the following keys to pop information inside the current window (if small enough) or into a help buffer. • The **<f12> 1** and **<f12> 2** PEL keys are available even when lispy mode is off. • In Common Lisp the slime (or SLY) mechanism is used to retrieved help information. 📦 The first 2 commands require the **lispy** external package. PEL downloads, installs and activates lispy when **pel-use-lispy** user option is set to **t**. 📦 All of the commands require one of **slime** or **sly** external package to be active. See PEL user-options at the top of this page. The commands used to start the Lisp REPLs are described after this block of commands, below.		
Describe function at point See also: Ψ- Lispy	**C-1**	(**lispy-describe-inline**)	Display documentation of current Lisp function: 'lispy--current-function' inline. • If docstring is small enough it is displayed in a pop-up box above point. Otherwise it is displayed inside a *lispy-help* buffer. • Hit the key sequence again to hide the pop-up box.
	<f12> 1		• The **<f12> 1** key can be used even when lispy mode is not active.
Describe function arguments See also: Ψ- Lispy	**C-2**	(**lispy-arglist-inline**)	Show the argument list of the function at point in a pop-up box. • Hit the key sequence again to hide the pop-up box.
	<f12> 2		• The **<f12> 2** key can be used even when lispy mode is not active.
Describe symbol at point	**C-c C-d C-d**	(**sly-describe-symbol** SYMBOL-NAME)	Describe the symbol at point in the *sly-description* buffer.
Describe function at point	**C-c C-d C-f**	(**sly-describe-function** SYMBOL-NAME)	Describe the function at point in the *sly-description* buffer. Includes the lambda-list, derived-type and source-form.
Symbol Apropos	**C-c C-d C-a**	(**sly-apropos** STRING &optional ONLY-EXTERNAL-P PACKAGE CASE-SENSITIVE-P)	Show all bound symbols whose names match STRING. • With prefix arg, you're interactively asked for parameters of the search. • With M-- (negative) prefix arg, prompt for package only.
Package Apropos	**C-c C-d C-p**	(**sly-apropos-package** PACKAGE &optional INTERNAL)	Show apropos listing for symbols in PACKAGE. • With prefix argument include internal symbols.
Apropos all	**C-c C-d C-z**	(**sly-apropos-all**)	Shortcut for (sly-apropos <string> nil nil)
• using Common Lisp Hyperspec ™	☝ Provides Standard Common Lisp topics, define, macros, etc. On-line and searchable. • The URL used for lookup is identified by the variable ***common-lisp-hyperspec-root***. • 🎛 With PEL, identify the location of the HyperSpec directory you want to use by writing it inside **pel-clisp-hyperspec-root** 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 ***common-lisp-hyperspec-root*** *variable.*		
Browse Common Lisp Hyperspec ™ reader macro	**C-c C-d #**	(**common-lisp-hyperspec-lookup-reader-macro** MACRO)	Browse the Common Lisp Hyperspec ™ entry for the reader-macro MACRO.
Lookup Common Lisp keywords in the Common Lisp Hyperspec	**<f12> ?**	(**pel-cl-hyperspec-lookup**)	Open Hyperspec documentation for symbol at point. Use the Slime, SLY or PEL mechanism, whatever is available.
	C-c C-d h	(**slime-documentation-lookup**)	Generalized documentation lookup. Opens a HyperSpec page. • Defaults to hyperspec lookup: opens a topic page in the browser Emacs uses.
	C-c C-d C-h	(**sly-documentation-lookup**)	Generalized documentation lookup. Defaults to **Hyperspec** lookup.
Look up Hyperspec Glossary	**C-c C-d C-g**	(**common-lisp-hyperspec-glossary-term** TERM)	View the definition of TERM on the **Common Lisp Hyperspec glossary** section.
• using Autodoc 1. With sly	SLY automatically shows information about symbols near the point with autodoc-mode, a SLY extension. • The informations shown on the echo area. For function names the argument list is displayed, and for global variables, the value. • Autodoc is implemented by means of eldoc-mode of Emacs.		
Toggle autodoc mode	**M-x sly-autodoc-mode**	(**sly-autodoc-mode** &optional ARG)	Toggle echo area display of Lisp objects at point. • Sly activates autodoc by default.
Explicitly request auto doc information	**M-x sly-arglist**	(**sly-arglist** NAME)	Show the argument list for NAME.
🚧	**M-x sly-autodoc-manually**	(**sly-autodoc-manually**)	Like sly-autodoc, but when called twice, or after sly-autodoc was already automatically called, display multiline arglist.
🚧	**C-c C-d A**	(**sly-autodoc** &optional FORCE-MULTILINE)	Returns the cached arglist information as string, or nil. • If it's not in the cache, the cache will be updated asynchronously.
• about REPL keys • See also: Σ Help/Info	Inside the REPL buffer window, use the **C-h m** (or **<f1> m**) key sequence to show the key sequences available inside the REPL. • The key sequences are differ across the inferior lisp, slime and SLY REPL. • 🚧 They are currently not documented here. What is documented are the key sequences available in the lisp-mode buffer while one of the REPL is available. • Slime has more key bindings in the REPL than sly, although sly has more available features (and they are accessible from the lisp-mode buffer). • Some of the common key bindings include: **<tab>** : completion of current input		
• Using Emacs' default *inferior-lisp* buffer 1. No slime or sly 2. With slime 3. With sly	The following commands can be used in 3 different setups: 1. When using Emacs **without** the **slime** or the **sly** external packages, **lisp-mode**'s **run-lisp** and **switch-to-lisp** commands open the basic *inferior-lisp* buffer which interact with your selected Common Lisp process. • The following commands are available from this basic *inferior-lisp* buffer and from the lisp-mode buffers once the *inferior-lisp* buffer is available. • They provide short cuts to execute Common Lisp code snippets in the Common Lisp REPL. However, the integration is not as nice as if you were running with the **slime** or the **sly** external packages activated: • The information queried is displayed directly inside the REPL, above your current line: a Common Lisp command is issued in the REPL. 2. With **slime** active 🚧 3. With **sly** active 🚧		
• Start the REPL • Switch to the REPL	The commands need access to a running Common Lisp REPL process. ☝ You must start the Common Lisp REPL using the following commands before executing commands to get help on code, evaluating, compiling and loading code as they all use the Common Lisp REPL.		
Run the appropriate REPL or switch to a window running it See also: Σ Shells	• **<f12> z**	(**pel-cl-repl** &optional N)	Open or switch to Common-Lisp REPL buffer window. Use the Common Lisp REPL selected by the PEL user-options: • SLY when `pel-used-sly' is on and `pel-clisp-ide' is set to sly, • Slime when `pel-use-slime'is on and `pel-clisp-ide' set to slime, the inferior lisp mode otherwise. The behaviour of the command is affected by the optional argument N: • with no buffers running REPL: • N is nil or absent: open REPL in current window • N is positive: open REPL in other window • N is negative: create new REPL in current window • with 1 or more REPL already running (if more than 1, prompt for one) • if selected buffer is inside an opened window: switch to that window • if selected buffer is not in an opened window: • N is nil or absent: open REPL in current window • N is positive: open REPL in other window • N is negative: create new REPL in current window.
	• **<f11> z r L**		
	M-x slime	(**slime** &optional COMMAND CODING-SYSTEM)	Start an inferior^_superior Lisp and connect to its Swank server.

Description	Keystroke	Function	Note

Start a Common Lisp REPL	`C-c C-z`	(**run-lisp** CMD)	If not already running, run an inferior Lisp process with I/O via '*inferior-lisp*' buffer.
			• If there is a process already running in '*inferior-lisp*', just switch to that buffer.
			• This runs the exterior program identified by the variable *inferior-lisp-program*.
			  • The PEL **pel-inferior-lisp-program** user-option controls it.
			  • By default, the value of this program is: "lisp".

		(**switch-to-lisp** EOB-P)	Switch to the inferior Lisp process buffer. The binding is done by run-lisp. With argument, positions cursor at end of buffer.
	`C-c C-z`	(**slime-switch-to-output-buffer**)	Select the output buffer, when possible in an existing window. ☝ Use 'display-buffer-reuse-frames' and 'special-display-buffer-names' to customize the frame in which the buffer should appear.
		(**sly-mrepl** &optional DISPLAY-ACTION)	Find or create the first useful REPL for the default connection. If supplied, DISPLAY-ACTION is called on the buffer. Interactively, DISPLAY-ACTION defaults to using 'switch-to-buffer' unless the intended buffer is already visible in some window, in which case that window is selected.
Sync SLY REPL	`C-c ~`	(**sly-mrepl-sync** &optional PACKAGE DIRECTORY EXPRESSION)	Go to the REPL, and set Slynk's PACKAGE and DIRECTORY. Also yank EXPRESSION into the prompt. Interactively gather PACKAGE and DIRECTORY these values from the current buffer, if available. In this scenario EXPRESSION is only set if a `C-u` prefix argument is given.
• **SLY Connections**			
Show all SLY connections	`C-c C-x c`	(**sly-list-connections**)	Display a list of all connections.
Switch to next SLY connection	`C-c C-x n`	(**sly-next-connection** ARG &optional DONT-WRAP)	Switch to the next SLY connection, cycling through all connections. • Skip ARG-1 connections. Negative ARG means cycle back. DONT-WRAP means don't wrap around when last connection is reached.
Switch to previous SLY connection	`C-c C-x p`	(**sly-prev-connection** ARG &optional DONT-WRAP)	Switch to the previous SLY connection, cycling through all connections. • See '**sly-next-connection**', above, for other args.
List SLY threads	`C-c C-x t`	(**sly-list-threads**)	Display the list of SLY threads.
• **Control Execution**			
Interrupt Lisp	`C-c C-b`	(**sly-interrupt**)	Interrupt Lisp. Impact depends on the Lisp implementation. SBCL presents the debugger backtrace prompt in a separate buffer from which you can select the next state.
• **Get Help on Code**	• Use the following commands to get information about the Common Lisp code.		
Show argument list	`C-c C-a`	(**lisp-show-arglist** FN)	Show the argument list of the defun/macro at point. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • It sends a query to the inferior Lisp for the arglist for function FN using the Common Lisp code identified by the variable 'lisp-arglist-command'.
Show symbol documentation	`C-c C-d`	(**lisp-describe-sym** SYM)	Send a command to the inferior Lisp to describe symbol SYM. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • Uses the command identified by variable 'lisp-describe-sym-command'.
Show function documentation	`C-c C-f`	(**lisp-show-function-documentation** FN)	Show docstring of function at point. Prompts, suggesting current function if any. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • Uses the command identified by variable 'lisp-function-doc-command'.
Show variable documentation	`C-c C-v`	(**lisp-show-variable-documentation** VAR)	Show documentation of variable at point. Prompts suggesting current variable if any. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • Uses the command identified by variable 'lisp-var-doc-command'.
• **Send code to the REPL**	• Use the following commands to send the Common Lisp code your are reading or writing to the REPL of the Common Lisp inferior process.		
• **Load Lisp code**	• Load existing Common Lisp files in the running REPL using the following command.		
Load Lisp file	`C-c C-l`	(**lisp-load-file** FILE-NAME)	Load a Lisp file into the inferior Lisp process. • Prompts with completion for the file to load. • Does not check for presence of file before passing it to the REPL.
	`C-c C-l`	(**slime-load-file** FILENAME)	Load the Lisp file FILENAME. • Use while point is in a source code buffer. Emacs prompt for the file name.
		(**sly-load-file** FILENAME)	Load the Lisp file FILENAME. This uses Common Lisp LOAD function.
Symbol Import/Export			
Export Symbol	`C-c x`	(**sly-export-symbol-at-point**)	Add the symbol at point to the defpackage source definition belonging to the current buffer-package. With prefix-arg, remove the symbol again. Additionally performs an EXPORT/UNEXPORT of the symbol in the Lisp image if possible.
Import Symbol	`C-c i`	(**sly-import-symbol-at-point**)	Add a qualified symbol to package's :import-from subclause. • Takes a package-qualified symbol at point, adds it to the current package's defpackage form (under its :import-form subclause) and replaces with a symbol name without the package designator.
• **Compile code, with:** 1. **No slime or sly** 2. **With slime** 3. **With sly**	• Use the following commands to compile Common Lisp code located in your buffer or in a file. ⚠ The compilation may harm **SLY Stickers** • Both Slime and SLY leave compiler annotations inside the source code buffer. The messages associated with the annotations can be read by playing the mouse over the text or with the commands described below.		
Compile current define form	`C-c C-c`	(**lisp-compile-defun** &optional AND-GO)	Compile the current defun in the inferior Lisp process. • DEFVAR forms reset the variables to the init values. • Prefix argument means switch to the Lisp buffer afterwards.
	`C-c C-c`	(**sly-compile-defun** &optional RAW-PREFIX-ARG)	Compile the current top-level form. • With positive (`C-u`) prefix argument: the form is compiled with maximal debug settings. • With negative prefix argument it is compiled for speed ('M--'). • With a numeric argument: set debug or speed settings to it depending on its sign.
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. • Creates a .fasl file in the directory holding the Common Lisp source file.
	`C-c C-k`	(**slime-compile-and-load-file** &optional POLICY)	Compile and load the buffer's file and highlight compiler notes. • With positive (`C-u`) prefix argument: the form is compiled with maximal debug settings. • With negative prefix argument it is compiled for speed ('M--'). • With a numeric argument: 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 'sly-next-note' and 'sly-previous-note' can be used to navigate between compiler notes and to display their full details.
		(**sly-compile-and-load-file** &optional POLICY)	
Compile a file (but don't load) current buffer	`C-c M-k`	(**sly-compile-file** &optional LOAD POLICY)	Compile current buffer's file and highlight resulting compiler notes. See 'sly-compile-and-load-file' for further details.
Compile the region	`M-x sly-compile-region`	(**sly-compile-region** START END)	Compile the region.
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.
		(**sly-next-note** N)	Go to and describe the next error button in the buffer. • Highlight the error in the *sly-compilation* buffer that describes the current detected problem and move point to the source cop the error.

Description	Keystroke	Function	Note
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.
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).
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.
		(**sly-previous-note** N)	Go to and describe the previous error button in the buffer. • Highlight the error in the *sly-compilation* buffer that describes the current detected problem and move point to the source cop the error.
Remove annotation notes	**C-c M-c**	(**sly-remove-notes** BEG END)	Remove 'sly-note' annotation buttons from BEG to END in the source code buffer. • The *sly-compilation* buffer is un-affected.
Disassemble	**C-c M-d**	(**slime-disassemble-symbol** SYMBOL-NAME)	Display the disassembly for SYMBOL-NAME. • 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.
	C-c M-d	(**sly-disassemble-symbol** SYMBOL-NAME)	Display the disassembly for SYMBOL-NAME.
• **Evaluate code, with:** 1. **No slime or sly** 2. **slime** 3. **sly**	Use the following commands to evaluate forms in a buffer that contains Common Lisp code and display the result in the echo area. • For commands using Slime, Slime must be active. It's the same for SLY.		
Evaluate last expression	**C-x C-e**	(**slime-eval-last-expression**)	Evaluate the expression preceding point. • Supports the eros-mode if installed.
	C-x C-e	(**sly-eval-last-expression**)	Evaluate the expression preceding point.
Evaluate current top-level form	• **C-c C-e** • **C-M-x**	(**lisp-eval-defun** &optional AND-GO)	Send the current form to the inferior Lisp process. • DEFVAR forms reset the variables to the init values. • Prefix argument means switch to the Lisp buffer afterwards.
	C-M-x	(**sly-eval-defun**)	Evaluate the current toplevel form. • Use 'sly-re-evaluate-defvar' if the from starts with '(defvar'
Evaluate expression typed in the mini buffer	• **C-c :** • **C-c C-e**	(**sly-interactive-eval** STRING)	Read and evaluate STRING and print value in minibuffer. A prefix argument('**C-u**') inserts the result into the current buffer. A negative prefix argument ('**M--**') will sends it to the kill ring.
Evaluate region	**C-c C-r**	(**lisp-eval-region** START END &optional AND-GO)	Send the current region to the inferior Lisp process. • ⚠️ A region must be marked, otherwise the REPL may go into debug mode. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • Prefix argument means switch to the Lisp buffer afterwards.
	C-c C-r	(**sly-eval-region** START END)	Evaluate region.
Eval paragraph	**C-c C-p**	(**lisp-eval-paragraph** &optional AND-GO)	Send the current paragraph to the inferior Lisp process. A paragraph is all forms between 2 sets of empty lines. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. • Prefix argument means switch to the Lisp buffer afterwards.
Evaluate expression before point & print result in fresh buffer	**C-c C-p**	(**sly-pprint-eval-last-expression**)	Evaluate the form before point; pprint the value in a buffer.
Edit setf-able value	**C-c E**	(**sly-edit-value** FORM-STRING)	Edit the value of a setf-able form in a new buffer '*Edit <form>*'. • The value is inserted into a temporary buffer for editing and then set in Lisp when committed with **C-c C-c**.
Undefine a function	**C-c C-u**	(**sly-undefine-function** SYMBOL-NAME)	Unbind the function slot of SYMBOL-NAME.
Evaluate last expression in Slime REPL	**C-c C-j**	(**slime-eval-last-expression-in-repl** PREFIX)	Evaluates last expression in the Slime REPL. • 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.
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. • Prints the information inside the *inferior-lisp* buffer, inside the REPL. ☛ This is also bound when slime is active.
Macro expansion with: 1. **slime** 2. **sly**	🚧 Expand macro expressions with the following commands		
Expand Macro form	**C-c C-m**	(**slime-expand-1** &optional REPEATEDLY)	Display the macro expansion of the form starting at point. • 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. • Inside the *slime-macro-expansion* buffer you can further expand with **C-c RET** and use (undo) to close the expansion.
		(**sly-expand-1** &optional REPEATEDLY)	Display the macro expansion of the form at point. • The form is expanded with CL:MACROEXPAND-1 or, if a prefix argument is given, with CL:MACROEXPAND. • Contrary to 'sly-macroexpand-1', if the form denotes a compiler macro, SLYNK-BACKEND:COMPILER-MACROEXPAND or SLYNK-BACKEND:COMPILER-MACROEXPAND-1 are used instead.
Macro expand expression	**M-x slime-macro-expand-1**	(**slime-macro-expand-1** &optional REPEATEDLY)	Macroexpand the expression starting at point once. If invoked with a prefix argument, use macroexpand instead of macroexpand-1.
Macro expand expression	**M-x sly-macro-expand-1**	(**sly-macroexpand-1** &optional REPEATEDLY)	Macroexpand the expression at point once. If invoked with a prefix argument, use macroexpand instead of macroexpand-1.
Expand macro form	**C-c M-m**	(**sly-macroexpand-all** &optional JUST-ONE)	Display the recursively macro expanded sexp at point. • With optional JUST-ONE prefix arg, use CL:MACROEXPAND-1.
Execute macro	**C-c e**	(**emacros-execute-named-macro**)	Prompts for the name of a macro and execute it. Does completion. • Default is the most recently saved, inserted, or manipulated macro in the current buffer.
Macro-expansion buffer commands with: 1. **slime** 2. **sly**	Extra commands available inside the macro-expansion buffer. 🚧		
Replace original form with macro-expansion	**C-c C-m**	(**sly-macroexpand-1-inplace** &optional REPEATEDLY)	Just like sly-macroexpand-1 but the original form is replaced with the expansion
Refresh macro expansion	**g**		The last macroexpansion is performed again, the current contents of the macroexpansion buffer are replaced with the new expansion.

Description	Keystroke	Function	Note
Close expansion buffer	`q`		Close the expansion buffer.
Undo last macro expansion	`C-_`		Undo last macroexpansion operation.
Introspection	who-calls-who is not yet implemented in CLisp.		
Show all specializations of class	• `C-c C-w a` • `C-c C-w C-a`	(**slime-who-specializes** SYMBOL)	Show all known methods specialized on class SYMBOL.
	`C-c C-w C-a`	(**sly-who-specializes** SYMBOL)	Show all known methods specialized on class SYMBOL.
Show all binders of global variable	• `C-c C-w b` • `C-c C-w C-b`	(**slime-who-binds** SYMBOL)	Show all known binders of the global variable SYMBOL.
	`C-c C-w C-b`	(**sly-who-binds** SYMBOL)	Show all known binders of the global variable SYMBOL.
Find who calls	• `C-c C-w c` • `C-c C-w C-c`	(**slime-who-calls** SYMBOL)	Show all known callers of the function SYMBOL.
	`C-c C-w C-c`	(**sly-who-calls** SYMBOL)	Show all known callers of the function SYMBOL. • This is implemented with special compiler support, see '**sly-list-callers**' for a portable alternative.
Show expanders of macro	• `C-c C-w m` • `C-c C-w RET`	(**slime-who-macroexpands** SYMBOL)	Show all known expanders of the macro SYMBOL.
	`C-c C-w RET` `C-c C-w C-m`	(**sly-who-macroexpands** SYMBOL)	Show all known expanders of the macro SYMBOL.
Show referrers of global variable	• `C-c C-w r` • `C-c C-w C-r`	(**slime-who-references** SYMBOL)	Show all known referrers of the global variable SYMBOL.
	`C-c C-w C-r`	(**sly-who-references** SYMBOL)	Show all known referrers of the global variable SYMBOL.
Show setters of global variable	• `C-c C-w s` • `C-c C-w C-s`	(**slime-who-sets** SYMBOL)	Show all known setters of the global variable SYMBOL.
	`C-c C-w C-s`	(**sly-who-sets** SYMBOL)	Show all known setters of the global variable SYMBOL.
Show functions called by	• `C-c C-w w` • `C-c C-w C-w`	(**slime-calls-who** SYMBOL)	Show all known functions called by the function SYMBOL.
	`C-c C-w C-w`	(**sly-calls-who** SYMBOL)	Show all known functions called by the function SYMBOL. • This is implemented with special compiler support and may not be supported by all implementations. • See '**sly-list-callees**' for a portable alternative.
List callers	`C-c <`	(**slime-list-callers** SYMBOL-NAME)	List the callers of SYMBOL-NAME in a xref window.
		(**sly-list-callers** SYMBOL-NAME)	List the callers of SYMBOL-NAME in a xref window. • See 'sly-who-calls' for an implementation-specific alternative.
List callees	`C-c >`	(**slime-list-callees** SYMBOL-NAME)	List the callees of SYMBOL-NAME in a xref window.
		(**sly-list-callees** SYMBOL-NAME)	List the callees of SYMBOL-NAME in a xref window. • See 'sly-calls-who' for an implementation-specific alternative.
Move to next location	`C-M-.`	(**slime-next-location**)	Go to the next location, depending on context. When displaying XREF information, this goes to the next reference.
Move to previous location	`C-M-,`	(**slime-previous-location**)	Go to the previous location, depending on context. When displaying XREF information, this goes to the previous reference.
Complete symbol at point	`C-c <tab>`	(**completion-at-point**)	Perform completion on the text around point. • 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)
Static Analysis			
Inspect expression	`C-c I`	(**slime-inspect** STRING)	Eval an expression and inspect the result.

	• Takes the expression at (or before) point, prompt to confirm it. On Return executes it and display result inside a *slime-inspector* buffer.
	  • It's often better to inspect the *symbol*, 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.
	  • Use `<f1> m` to list all available commands and their key bindings. These include:
	  • `RET` : Inspect item at point
	  • `l` : pop-up inspection level		

	`C-c I`	(**sly-inspect** STRING &optional INSPECTOR-NAME)	Eval an expression and inspect the result.
Debugging	The following commands help debug Common Lisp code. - These work under GNU CLisp		
Show Trace Dialog	`C-c T`	(**sly-trace-dialog** &optional CLEAR-AND-FETCH)	Show trace dialog and refresh trace collection status. • With optional CLEAR-AND-FETCH prefix arg, clear the current tree and fetch a first batch of traces.
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.
		(**sly-trace-dialog-toggle-trace** &optional USING-CONTEXT-P)	Toggle the dialog-trace of the spec at point. • When USING-CONTEXT-P, attempt to decipher lambdas. methods and other complicated function spec
	`C-c M-t`	(**sly-toggle-fancy-trace** &optional USING-CONTEXT-P)	Toggle trace.
Sly Stickers			
Forget sly stickers	`C-c C-s F`	(**sly-stickers-forget** &optional HOWMANY INTERACTIVE)	Forget about sticker recordings in the Slynk side. • If HOWMANY is non-nil it must be a number stating how many recordings to forget about. In this cases Because 0 is an index, in the 'nth' sense, the HOWMANYth recording survives.
Fetch update sly stickers	`C-c C-s S`	(**sly-stickers-fetch**)	Fetch recordings from Slynk and update stickers accordingly. See also '**sly-stickers-replay**'.
Clear sly stickers in current top level form	`C-c C-s C-d`	(**sly-stickers-clear-defun-stickers**)	Clear all stickers in the current top-level form.
Clear sly stickers in current buffer	`C-c C-s C-k`	(**sly-stickers-clear-buffer-stickers**)	Clear all the stickers in the current buffer.

Description	Keystroke	Function	Note
Interactive reply	`C-c C-s C-r`	(**sly-stickers-replay**)	Start interactive replaying of known sticker recordings.

Set/Remove sly sticker at point	`C-c C-s C-s`	(**sly-stickers-dwim** PREFIX)	Set or remove stickers at point.
			• Set a sticker for the current sexp at point, or delete it if it already exists.
			• If the region is active set a sticker in the current region.
			• With interactive prefix arg PREFIX always delete stickers.
			  • One C-u means delete the current top-level form's stickers.
			  • Two C-u's means delete the current buffer's stickers

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 '𝕊'). 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: • 𝕊 **Cut & Paste** • (CLKB sl2.lisp)	• **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: • 𝕊 **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. 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> - [**
	⚠ 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.		
Kill Lisp S-Expression at point See also: 𝕊 **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: 𝕊 **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.
mark sexp and balanced expressions See also: • 𝕊 **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: 𝕊 **Marking**	• **M-=** • **<f11> . =**	(**er/expand-region** ARG)	Increase selected region by semantic units. With prefix argument: • positive number: expands the region that many times. • negative: calls 'er/contract-region'. • 0: resets point & 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. • 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. 📦 This requires the **expand-region** package. Under PEL, activated with ***pel-use-expand-region*** user option. ☝ 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.		

Description	Keystroke	Function	Note
Navigation in LISP	This current list below describe the specialized commands only. See the others inside **Navigation**		
• **Using iMenu** See also: • **Menus** • **Navigation**	PEL provides access to several commands that use the iMenu system to parse Common Lisp code buffers and show lists of Common Lisp definitions in various ways: completion buffers, ido, ivy or helm lists as well as popup menu or specialized popup menu. Several commands are available and are detailed in the **Navigation** and the **Menus** pages. This includes the **M-g i**, **M-g h** and **M-g y** key sequences as well as the **M-g <f4>** key prefix for controlling their behaviour. For Common Lisp the following extra key binding is available:		
Add a new symbol to iMenu parsing	**M-g <f4> .**	(**pel-cl-add-symbol-to-imenu**)	Add symbol at point to imenu.

	Common Lisp macro can define code definition forms similar to 'defun' and friends, effectively creating a Domain Specific Language. The DSL symbols may not currently be know to 'imenu' parsing. You can add new symbols to 'imenu' by placing the point over such a symbol and executing this command. For example, if the file's code uses a 'define-rule' macro like this:
	`(define-rule rule1`
	`  (do-this)`
	`  (do-that)`
	`  (ensure this and that))`
	`(define-rule secondary`
	`  (ensure something-else))`
	Place point over 'define-rule' and execute the command. You will be prompt for a title for 'define-rule', where you could enter "rules". Then the 'imenu' list will be able to show the "rule" and "secondary" under the "Rules" iMenu section.		

• **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. • If there's no name at point, or a prefix argument is given, prompt for function name.
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…
	The following `**beginning-of-defun**' and `**end-of-defun**' are standard Emacs commands. They have limitations:
	• They only navigate across any top-level form.
	  • 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:
	  • **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)	• **<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.
			• It toggles the standard Emacs `**open-paren-in-column-0-is-defun-start**' between:
			  • 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 defun start.

Backward to beginning of defun See also: **Navigation**	• **C-M-a** • **C-M-<home>** • **<f6> <up>**	(**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>**). It's always available for **<f6> <up>** : hold Shift after typing **<f6>**.
	☝ By default Emacs treats all opening parenthesis character in the first column as a defun. • 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. • 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**. ⚠ Moves to beginning of next function of the same nesting level of the current location. Skips the functions and methods that are more deeply nested.		
Forward to end of defun See also: **Navigation**	• **<f12> <right>** • **M-<f12> <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 (both keys). However **<f6> <right>** and **<f12> <right>** handle Shift-marking fine in terminal mode. ⚠ This command moves to the end of the next **top-level** function or class.
	• **C-M-e** • **C-M-<end>** • **<f6> <right>**		
Forward to start of next defun	**<f6> <down>**	(**pel-beginning-of-next-defun** ARG)	Move forward to the beginning of the next top-level form: function definition, macros, etc.. • 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-`** or **<f6><f6>**. ☛ Shift marking is available with **<f6> <down>** : hold Shift after typing **<f6>**.
	☝ This command is generic and for Emacs Lisp, moves to the beginning of the next top-level form. • Complements what end-of-defun does. Moves forward 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. • 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. • 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	• **<f12> <left>** • **M-<f12> <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-`** or **<f6><f6>**. ☛ Shift marking is available.
	<f6> <left>		

• **To next/previous selected top-level form or defun or …**
★★
	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…
	☝ PEL provides the following powerful commands: **pel-elisp-beginning-of-next-form** and **pel-elisp-beginning-of-previous-forms**.
	  • 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.
	☝ PEL also provides specialized versions of these commands:
	• **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 *name* 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 *name* of the next form, **pel-elisp-to-name-of-previous-form** to the previous one.		

Change target form for commands:
• **<f12> <up>**
• **<f12> <down>**
• **<f12> C-<up>**
• **<f12> C-<down>**
★★
	• **<f12> M-n**
	• **M-<f12> M-n**
		(**pel-elisp-set-navigate-target-form** &optional GLOBALLY)	Select form navigation behaviour.
			Select the behaviour of the following navigation functions:
			  • 'pel-elisp-beginning-of-next-form' and
			  • 'pel-elisp-beginning-of-previous-form'.

	<f11> SPC L M-n		• 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.

Description	Keystroke	Function	Note
Forward to start of next definition form ★★ **Configurable target:** • **all top-level forms** • **top-level defun** • **all defun** • **all defun, defsubst, defmacros, …** • **all variable definition forms: defvar, defconst, defcustom, defgroup, …** • **etc…**	• **<f12> <down>** • **M-<f12> <down>** **<f11> SPC L <down>**	(**pel-elisp-beginning-of-next-form** &optional N TARGET SILENT DONT-PUSH-MARK)	Move point forward to the beginning of next N top-level form. • 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**'.
	• 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-`** or **<f6><f6>**. ☛ Shift marking is available with **<f12> <down>** ☝ This command is the most flexible and can be configured to move like the next 2 commands. • 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. • 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: • 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	• **<f12> C-<down>** • **M-<f12> C-<down>**	(**pel-elisp-to-name-of-next-form** &optional N)	Move point to the name of next N defun form - at any level. • Skip over forms located inside docstrings. Leave point on the first character of the form name. • Move back to previous position with **M-`** or **<f6><f6>**.
Forward to beginning of next defun form	• **<f12> M-<down>** • **<f12> f n** • **M-<f12> f n** **<f11> SPC L f n**	(**pel-elisp-beginning-of-next-defun** &optional N)	Move point to the name of the next defun form, whether it is top-level or indented. • The function skips over forms inside docstrings. • Move back to previous position with **M-`** or **<f6><f6>**. • 💻 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	• **<f12> C-<M-down>** • **M-<f12> C-<M-down>**	(**pel-elisp-to-name-of-next-defun** &optional N)	Move point to the name of next N defun form - at any level. • 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-`** or **<f6><f6>**.
Backward to start of previous definition form ★★ **Configurable target:** • **all top-level forms** • **top-level defun** • **all defun** • **all defun, defsubst, defmacros, …** • **all variable definition forms: defvar, defconst, defcustom, defgroup, …** • **etc…**	• **<f12> <up>** • **M-<f12> <up>** **<f11> SPC L <up>**	(**pel-elisp-beginning-of-previous-form** &optional N TARGET SILENT DONT-PUSH-MARK)	Move point backward to the beginning of previous N top-level form. • 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-`** or **<f6><f6>**. ☛ Shift marking is available **<f12> <up>**
	☝ This command is the most flexible and can be configured to move like the next 2 commands. • 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. • 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: • 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.		
Backward to the name of the previous form definition	• **<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. • Skip over forms located inside docstrings. Leave point on the first character of the form name. • Move back to previous position with **M-`** or **<f6><f6>**.
Backward to beginning of previous defun form	• **<f12> M-<up>** • **<f12> f p** • **M-<f12> f p** • **<f11> SPC L 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. • 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-`** or **<f6><f6>**. • 💻 Uses **pel-elisp-beginning-of-previous-form** specifying 'defun-forms as target type. ☛ Shift marking is available with **<f12> M-<up>**
Backward to the name of the previous defun definition	• **<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. • 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-`** or **<f6><f6>**.
• **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: 𝕀 **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.
Move block backward See also: • 𝕀 **Navigation**	• **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). • 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.

Description	Keystroke	Function	Note
	⚠ 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: Σ **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.
Move block forward See also: • Σ **Navigation**	• **`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). • 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: • Σ **Navigation**	• **`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. • 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: Σ **Navigation**	**`C-M-]`**	(**up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward 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 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: • Σ **Navigation**	• **`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. • 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: • Σ **Text Modes** • Σ **Search/Replace**	• **`<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. • 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…)
SemEd - Transpose			
Transpose two balanced expressions (sexps) See also: Σ **Transpose**	• **`C-M-t`** • **`<f11> t t x`**	(**transpose-sexps** ARG)	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 *sexps*.
	• 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 - Code Indentation with: 1. **No slime or sly** 2. **With slime** 3. **With sly**	Lisp code indentation is governed by the Common Lisp language and depend on the various S-Expression forms. ☝ **Indentation of Lisp Code** As opposed to most programming languages, in Lisp family languages there is a strong relationship between the indentation and the code validity when indentation is done automatically by tools like Emacs. The editor is able to detect what should be the indentation by looking at each S-expression. Emacs will be able to indent the code based on the Lisp forms using the commands below. If the indentation is wrong, there's a very good chance that your code is not doing what you think it should do! To check Lisp code, don't look at end parenthesis: look at the indentation. When editing use the provided tools to indent code automatically. Learn to trust Emacs for indentation. Don't fight it. • 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 Common Lisp the function to use is common-lisp-indent-function . • The slime-setup function adds the slime-lisp-mode-hook function to the lisp-mode-hook. • The slime-lisp-mode runs the required following code to install the indenter for Common Lisp: `(set (make-local-variable lisp-indent-function) 'common-lisp-indent-function)` When Slime or SLY are used, the Lisp code is scanned for macros that have &body arguments so code using these macros can be indented properly. See the details in the links identified in the title column.		
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 (CLBC s3.lisp)	**`C-M-q`**	(**indent-sexp** &optional ENDPOS)	Indent each line of the S-expression starting just after point. • If optional arg ENDPOS is given, indent each line, stopping when ENDPOS is encountered.

Description	Keystroke	Function	Note
SemEd - Parentheses	The commands below are used to help dealing with the parentheses (along with the semantic editing navigation commands listed above). ☛ To insert and move through parentheses you can also use **λ- Lispy**. It provides even simpler key strokes.		
Insert Parentheses (See also: • **λ - Emacs 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 **λ - Emacs 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.
Check validity of parentheses **(or quotes, braces, brackets)** See also **λ - Emacs Lisp**	• **<f12>)** • **M-<f12>)** • **<f11> SPC L)**	(**check-parens**)	Check for unbalanced parentheses (or quotes, braces and brackets) 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.
Close all parentheses of open expression at point	**C-c C-]**	(**slime-close-all-parens-in-sexp** &optional REGION)	Balance parentheses of open s-expressions at point. • 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. • 🚧 *I noticed that it does not always work: need to investigate)*
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. You can also use Graphviz, see **Graphviz Dot**		

Preview UML diagram from plantUML source in current plantUML region of commented source code
 	
See also: **PlantUML**
	<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
			📦 Requires the **plantuml-mode** external package, activated by **pel-use-plantuml**.

	☝ 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.		
Common Lisp Tools	The following sections describe some of the open source tools available for Common Lisp development.		
• **Quicklisp**	• **Installation instructions**: start here. Next lines assume Quicklisp is installed.		
Load Quicklisp manually in REPL	Load Quicklisp	Execute the following inside the REPL: **(load "~/quicklisp/setup.lisp")**	
Configure Common Lisp to automatically load Quicklisp	Configure Common Lisp to automatically load quicklisp	• Execute the following inside the REPL: **(ql:add-to-init-file)** • It will update the RC fie used by your compiler, like ~/.sbclrc used by SBCL	
List what is available in Quicklisp	Use a sub-string, like:	**(ql:system-apropos "xml")**	
Load a Common Lisp package	Example: vecto	**(ql:quickload "vecto")**	
Remove a package	Example: vecto	**(ql:uninstall "vecto")**	
To update a package	Example: quicklisp itself:	**(ql:update-dist "quicklisp")**	
To update Quicklisp client	Quicklisp client updates are available a few times per year	**(ql:update-client)**	
See: Going Back in (dist) time			
List dependencies	Example: vecto	**(ql:who-depends-on "vecto")**	

Common Lisp Support — References

Description & URL	Notes
Lisp	
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 ANSI Standard — INCITS 226-1994 (R1999) (formerly ANSI X3.226-1994)	The Common Lisp standard.
Common Lisp Community Spec (CLCS)	A separate rendering of the Common Lisp Specs, a more modern interface that appeared recently, based on the original specification documents. Very nice. A left side bar with hierarchical table of content and search field helps navigation.
Common Lisp Nova Spec	Another rendering/search engine for the Common Lisp language. Also appeared recently (2023) and very nice. Can be read like a book, one page at a time with previous/next links, table of content, and search navigation.
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. • The Wikipedia page for the Common Lisp HyperSpec provides links to the main page as well as the set of page data.
Common Lisp Books & Tutorials	The following pages contain links to several books on Common Lisp and related subjects: • lisp-lang.org Common Lisp Books • Wikipedia Common Lisp Publications
The Common Lisp Cookbook	This is a online 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.
Programming Algorithms in Lisp, 2021	Writing Efficient Programs with Examples in ANSI Common Lisp, Vsevolod Domkin, 2021

Description & URL	Notes
ANSI Common Lisp - by Paul Graham, 1995	
On Lisp: Advanced Techniques for Common Lisp - by Paul Graham	An important book to learn Common Lisp and how to write elegant macros. • On Lisp, unofficial HTML 2002 version @ wayback machine • On Lisp @ CLiki the Common Lisp wiki. has more information wit hills to a .texi file.
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 Quick Reference	A Latex-based quick reference, written by Bert Burgemeister. With several PDF rendering, 2 formats ready for printing, one for on-screen reading. • Source on trebb/clqr Github project
Object-Oriented Programming in Common Lisp - Sonja E. Keene	A Programmer's Guide to CLOS. (Common Lisp Object System). The book is in print and difficult to get. • This quick overview of the Common LISP Object SYSTEM gives a starting point. • Object Orientation tutorial @ lisp-lang.org
Successful Lisp: How to Understand and Use Common Lisp	From David B. Lamkins. <dlamkins@psg.com> hosted @ Collège Sciences et Technologies / Université de Bordeaux.
CLOG - Common Lisp Omniscient GUI	• Learning CLOG - a tutorial for learning Common LISP through CLOG, a system for building GUI programs. • CLOG @ Github
Macros	
Anaphoric macros @ Let over lambda	
Toward Fearless Macros	Interesting article about macros in several programming languages including Lisp. It's a bit weak on the C preprocessor macro description, but the rest of the document has some value.
Common Lisp Resources	**CLiki - Common Lisp wiki**
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. • 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 Community	
Reddit - Common Lisp	The r/Common_Lisp top page has a side bar with useful information.
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) • 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. • 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	An older version of the above page, 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. • It uses a backend Common Lisp compiler that must be installed separately and identified by the ***inferior-lisp-program*** variable, and which is tied to the Emacs buffer identified by the ***inferior-lisp-buffer*** 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, ADSF and Quicklisp. At this point in time (Jan 2020), it seems that it's better to install them separately.
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.
Common Lisp Tools	• On new IDEs (blog)

Description & URL	Notes	
Library Manager: Quicklisp	A library manager for Common Lisp. The site describes Quicklisp installation and use. Also see the library list and Quicklisp FAQ which describes how to add a library to Quicklisp, among other things. There's also links to several related blog posts: • Getting a library into Quicklisp • Some problems when adding libraries to Quicklisp • Quicklist blog archive describes the newly added libraries • **Quicklisp-projects @ GitHub**. The projects sub-directory contains all information about the various libraries, including the URL where the files are taken.	
Quickref : Reference manuals for Quicklisp Libraries	A list of the Common Lisp libraries supported by QuickLisp, with links to the documentation of each of those. The libraries are indexed by name and author names. Unfortunately the URLs of the libraries repos are not included. But that can be identified as described in the cell above.	
Build Control - ASDF	ASDF := Another System Definition Facility. A package format DSL (.asd files) and a build tool. • ASDF home page • ASDF Manual • Getting started with ASDF • tychoish blog: Common Lisp, Using ASDF Install With SBCL	
Roswell - Common Lisp environment setup Utility	A command line tool used to setup Common Lisp environment. • Roswell home @ GitHub • Roswell code @ Github	
Common Lisp Libraries		
Alexandria	• Alexandria @ Gitlab • Alexandria Manual	
Common Lisp Portability Status	Quick table listing libraries & their support by Common Lisp implementation with links to the libraries.	
Awesome Common Lisp	A curated list of *awesome* Common Lisp libraries - a GitHub hosted page.	
Trending Common-Lisp @ GitHub	Popular GitHub Common Lisp projects	
Common Lisp repos @ GitHub	Using GitHub's Advanced Search. Use extra filter criteria for more precise results.	
• **Parallelism & Concurrency Libraries**	• The Common Lisp Cookbook – Threads, concurrency, parallelism. A good introduction the topic.	
Multithread library: Bordeaux Thread	• bordeaux-threads @ GitHub • Bordeaux-Threads official documentation • Brodeaux-Thread Blog	• Bordeaux-Thread API V2 blog entry • Discussion on Bordeaux-Thread API v2 on reddit • Bordeaux Thread Portable shared-state concurrency for Common Lisp (a little old) @ quicklisp
lparallel : a library for parallel programming	• lparallel @ GitHub • lparallel documentation	
Papers/Presentations on/about Lisp		
Lisp: Where do we come from? Where are we? Where are we going?	Lisp User Group presentation, October 11, 1999, by Peter Norvig from http://norvig.com	
Design Patterns in Dynamic Languages	Another interesting presentation, circa 1996, by Peter Norvig	