↖ Topic Index

Emacs support for the Erlang Programming Language

Description	Keystroke	Function	Note
Erlang Support See also: • **Erlang Reference** • **PEL Manual** • **about-erlang** • **Developing Erlang Code with PEL** • set PEL Erlang environment • **∑ Hide/Show** • **∑ Text Modes** • **∑ Highlight** • **∑ Inserting Text** • **∑ Speedbar** • **∑ Customize** ☝ ➽ **Identify minor modes to activate automatically in erlang-mode buffers** *Erlang Style Control* ☛ • **Ericsson AB Guideline** • **Inaka Guideline**			Emacs supports Erlang via the following external packages: 📦 The **erlang.el** external package (see erlang.el source), part of **OTP** ↗ PEL activates it when **pel-use-erlang** is turned on. It can then also activates: 📦 The **EDTS** external package ↗ activated by **pel-use-edts** (set to **t** or **start-automatically**). 📦 The **lsp-mode** external package ↗ activated by **pel-use-erlang-ls**. Uses the **erlang_ls** Erlang LSP server. Integrates with: • 📦 **Helm** by using **helm-lsp** ↗ activated by **pel-use-helm-lsp**. 📦 **Ivy** by using **lsp-ivy** ↗ activated by **pel-use-lsp-ivy**. • 📦 **treemacs** by using **lsp-treemacs** ↗ activated by **pel-use-treemacs** and **pel-use-lsp-treemacs**. • 📦 **origami** by using **lsp-origami** ↗ activated by **pel-use-lsp-origami**. 📦 The **flycheck** external package ↗ activated by **pel-use-erlang-syntax-check** set to 'use-flycheck, or Emacs built-in **flymake** if set to 'use-flymake. The **Distel** external package also exists, but seems to have mainly been replaced by EDTS and needs maintenance. PEL does not support it. 📦 The **hide-comnt.el** external package ↗ activated by **pel-use-hide-comnt** 📦 The **iedit** external package ↗ activated by **pel-use-iedit**. 📦 The **smart-dash** external package ↗ activated by **pel-use-smart-dash**. 📦 The **smartparens** external package ↗ activated by **pel-use-smartparens**. ☛Activate smart-dash-mode or smartparens-mode automatically in erlang-mode buffers by adding their mode to **pel-erlang-activates-minor-modes**. ☛Add electric pairing without smartparens with built-in **electric-pair-local-mode**: add electric-pair-local-mode to **pel-erlang-activates-minor-modes** . • Use **<f12> <f3> electricity RET** to access the customization group and select pairs. ☛Useful global minor-modes to activate features in Erlang via **pel-activates-global-minor-mode**: **show-paren-mode** ☛ PEL adds **∑ Speedbar** for .erl, .hrl and .escript Erlang files to show the list of functions. 💻 PEL Erlang support implemented in: **pel-erlang.el**, **pel-erlang-skels.el**, sections of **pel--key-macros.el** and **pel_keys.el** and PEL files they require. 🎛 **Customization:** • Type **<f11> <f2> g** followed by the group name and **RET** to open the specific customization group or one of the following key sequences. • **pel-pkg-for-erlang**: to activate **pel-use-erlang**: use **<f11> SPC e <f2>** , or **<f12> <f2>** from an Erlang buffer. This has sub-group: see **pel-erlang-ide** group to activate EDTS and LSP. • **erlang**: when **pel-use-erlang** is on, use **<f11> SPC e <f3> 1** • **edts**: when **pel-use-edts** is on, use **<f11> SPC e <f3> 3** • **lsp-erlang**: when **pel-use-erlang-ls** is on, use **<f11> SPC e L <f3> 1** • **lsp-mode**: when **pel-use-erlang-ls** is on, use **<f11> SPC e L <f3> 2** The **pel-pkg-for-erlang** group has several user-options to control Erlang editing. Only some of them are described here. Use Emacs for the complete list. • **pel-erlang-shell-prevent-echo:** set to **t** to prevent the Erlang shell from echoing every command. • **pel-erlang-activates-minor-modes**: Schedules activation of local minor modes in erlang-mode buffers, eg. smart-dash-mode. • **pel-erlang-environment group**: • **pel-erlang-man-parent-rootdir:** Identifies the parent directory of Erlang man directory. The man directory should hold the man1, man3, man4 and man6 which contain Erlang man files. If this is set PEL sets (override) the **erlang.el** **erlang-root-dir** user-option value with it which activates the appropriate Erlang man files. Without PEL or if pel-erlang-man-parent-rootdir is nil, you must set the erlang-root-dir user-option yourself. • **pel-erlang-exec-path**: Identifies the directory where Erlang binaries are stored. • **pel-erlang-version-detection-method**: identifies a mechanism to detect Erlang/OTP version. By default it uses an Erlang script provided with PEL. • **pel-erlang-code-style group**: • **pel-erlang-fill-column** : column where line-wrapping occurs : maximum ***line length*** (defaults to 100). You can change the value or set it nil. • When pel-erlang-fill-column user option is nil, erlang-mode buffers use the global Emacs fill-column value. • **pel-erlang-skel-use-separators**: whether line separators are used in Erlang code templates (see the *Insert Erlang Code Template* section below), • **pel-erlang-skel-use-secondary-separators** : whether secondary separator lines are inserted by some Erlang code templates, • **pel-erlang-skel-insert-file-timestamp**: whether automatically updated time stamps are inserted in Erlang source code file header blocks. • **pel-erlang-space-after-comma-in-blocks**: when turned on, a space is automatically inserted after a comma typed inside a parens block.
Open this PDF file. See also: **∑ Help/Info**	• **<f11> SPC e <f1>** • **<f11> SPC e w <f1>** • **<f11> SPC e L <f1>** • **<f12> <f1>** • **<f12> w <f1>** • **<f12> L <f1>**	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **𝔅ĭ - Erlang** 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. ☝Key sequences that start with **<f11> SPC e** are available from any major modes. Key sequences that start with **<f12>** are only available in erlang-mode buffers. The **<f12>** keys sequences are mirrored by the **<M-f12>** key sequence for convenience.
∑ Customize PEL Erlang support	**<f11> SPC e <f2>** **<f12> <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL Erlang support: access PEL user-options to activate Erlang support packages. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
∑ Customize Emacs Erlang support	**<f11> SPC e <f3>** **<f12> <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs Erlang support: erlang, erldoc, erlstack, edts, ivy-erlang-complete, lsp-erlang, lsp-mode, lsp-treemacs, auto-highlight-symbol, electricity, smart-dash, smartparens, treemacs. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
∑ Customize PEL LSP for Erlang support	**<f11> SPC e L <f2>** **<f12> L <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL LSP Erlang support • If OTHER-WINDOW is non-nil (use **C-u**), display in another window. ↗ This is available when **pel-use-erlang-ls** is turned on.
∑ Customize Emacs LSP for Erlang support	**<f11> SPC e L <f3>** **<f12> L <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs LSP Erlang support: lsp-erlang, lsp-mode, lsp-ui, helm-lsp, lsp-ivy, lsp-origami, lsp-treemacs. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window. ↗ This is available when **pel-use-erlang-ls** is turned on.
∑ Customize PEL LSP Window for Erlang support	**<f11> SPC e w <f2>** **<f12> w <f2>**	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL LSP Erlang support • If OTHER-WINDOW is non-nil (use **C-u**), display in another window. ↗ This is available when **pel-use-treemacs** and/or **pel-use-lsp-treemacs** is turned on.
∑ Customize Emacs LSP Window for Erlang support	**<f11> SPC e w <f3>** **<f12> w <f3>**	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs LSP Erlang support: lsp-treemacs, treemacs • If OTHER-WINDOW is non-nil (use **C-u**), display in another window. ↗ This is available when **pel-use-treemacs** and/or **pel-use-lsp-treemacs** is turned on.
Environment Help	Use the following command to verify your Erlang environment.		
Erlang Mode version	**<f11> SPC e ?**	(**pel-show-erlang-version**)	Display information about Erlang and Emacs Erlang supporting tools in the echo area. This includes the version of Erlang, erlang_ls, ivy-erlang-complete, the Erlang root path and its detection method, directory for Man files, lsp-keymap-prefix, etc…
	<f12> ?	Displays current version of available Erlang system, of **erlang.el**, of **erlang_ls** (if available), values of erlang-root-dir and pel-erlang-man-parent-rootdir. ⚠Check that **erlang-root-dir** matches the version of Erlang you use. If not check the setting of the **erlang-man-parent-rootdir**. For more information see **set PEL Erlang environment**.	
Syntax Highlighting	The **erlang.el** external package provides several levels of Erlang code syntax highlighting: • Off, Level 1: comments only, Level 2, Level 3, Level 4: maximum variety. There is not key binding for this. You must use the Syntax Highlighting section of the Erlang menu: • In terminal mode Type **<f10>** to access the menu, then select Erlang, Syntax Highlighting and the level you want.		

Description	Keystroke	Function	Note
Electric Keys for Erlang ∑ Customize • 🎛 electric keys	Two different packages have an impact on the "***electric***" behaviour of some keys in erlang-mode buffers: 1. the **erlang.el** external package, which controls the behaviour of the **RET**, **,**, **;** and **>** keys as controlled by **erlang-electric-commands** variable. 2. the **smartparens** external package, which modifies the behaviour of the **DEL** and **<deletechar>** behaviour when **smartparens-mode** is active. • Use **<f11> (** to toggle smartparens-mode on and off. PEL provides customization and dynamic control of erlang.el electric key behaviour and provides electric behaviour of some extra keys. • The **pel-erlang-electric-keys** user-option set which of the **RET**, **,**, **;** and **>** keys have electric behaviour. By default they are all activated. • The **pel-erlang-space-after-comma-in-block** user-option activates automatic insertion of space after comma inside a block. Disabled by default. • Inside an erlang-mode buffer, use the **<M-f12> M-`** prefix key followed by one of these keys to toggle the electric behaviour of the key.		
Toggle , electricity	**<M-f12> M-` ,**	(**pel-erlang-comma** &optional GLOBALLY)	Toggle electric behaviour of the comma key. Show message describing its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` ,**
Toggle automatic insertion of space after comma in block	**<M-f12> M-` M-,**	(**pel-erlang-toggle-space-after-comma** &optional GLOBALLY)	Toggle automatic insertion of space after comma inside blocks. Show its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` M-,**
Toggle > electricity	**<M-f12> M-` >**	(**pel-erlang-gt** &optional GLOBALLY)	Toggle electric behaviour of the greater-than key. Show message describing its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` >**
Toggle RET electricity	**<M-f12> M-` RET**	(**pel-erlang-newline** &optional GLOBALLY)	Toggle electric behaviour of the newline key. Show message describing its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` RET**
Toggle ; electricity	**<M-f12> M-` ;**	(**pel-erlang-semicolon** &optional GLOBALLY)	Toggle electric behaviour of the semicolon key. Show message describing its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` ;**
Toggle . electricity	**<M-f12> M-` .**	(**pel-erlang-period** &optional GLOBALLY)	Toggle Erlang electric behaviour of the semicolon key. Show message describing its new state. • To modify the behaviour in all Erlang buffers type: **M-- <M-f12> M-` .**
Toggle - electricity	**<M-f12> M-` -**	(**smart-dash-mode** &optional ARG)	Toggle the smart-dash-mode on/off. More info in ∑ **Text Modes** and ∑ **Inserting Text** .
Matching Pairs	By default the erlang-mode syntax table defines matching pairs made of **( )**, **[ ]**, **{ }**, **" "** and **' '**. PEL adds the **<< >>** pair. • With smartparens-mode activated typing the opening character(s) automatically inserts the closing character(s) 📦 This requires **smartparens** external package. 🔄 activated by **pel-use-smartparens**. Add smartparens-mode to **pel-erlang-activates-minor-modes** to activate smartparens-mode automatically for erlang-mode buffers. ☝ Add electric pairing without smartparens with built-in **electric-pair-local-mode**: add electric-pair-local-mode to **pel-activates-minor-modes** list.		
Matching pairs • ∑𝑥 Smartparens	**(** **[** **{** **"** **'** **<<**	When the **smartparens** external package is used and the smartparens-mode is active, the characters on the left are taken to be part of a pair. The pairs are: **( )**, **[ ]**, **{ }**, **" "**, **' '**, and **<< >>** (added by PEL). • When typing the first character of a pair, the rest of the pair is inserted and point is left inside. • To enclose a piece of text inside one of those pairs, mark the text area then type the first character of the pair. • The smartparens-mode can be activated automatically for Erlang by adding erlang-mode to the **pel-erlang-activates-minor-modes** user-option. • Use the **<f11> (** key sequence to toggle the smartparens-mode on and off. • There's also the smartparens-strict-mode that imposes balanced pairs but that does not help much in Erlang. • PEL adds support for **<< >>** including navigation across balanced pairs, something the default erlang.el does not do, by replacing forward-sexp and backward-sexp by specialized functions.	
Insert Parentheses	**M-(**	(**insert-parentheses** &optional ARG)	For Erlang: insert a parenthesis pair '()', leaving point after open-paren. Use this when smartparens is not used.
	• A positive ARG encloses the following ARG sexps in parenthesis if they are balanced. A negative ARG encloses the preceding ARG sexps instead. • No argument is equivalent to zero: just insert '()' and leave point between. If region is active, insert enclosing characters at region boundaries. • PEL makes 'parens-require-spaces' buffer local and set it to nil in Erlang mode buffers, allowing the use of this command to insert the argument parentheses following a function (and without placing a space between the function name and the opening parenthesis.		
New Line *Electric behaviour:* • *indent next line*	**RET**	(**erlang-electric-newline** &optional ARG)	Break line at point. If electric behaviour is activated: indent, continuing comment if within one. • Should the current line begin with a comment, and the variable 'comment-multi-line' be non-nil, a new comment start is inserted. • Should the previous command be another electric command we assume that the user pressed newline out of old habit, hence we will do nothing.
			The *electric* behaviour of this key is controlled by 2 variables: • **erlang-electric-commands** must include the **erlang-electric-newline** symbol to activate the key electric behaviour. • **erlang-electric-newline-criteria** identifies how to check whether newline should behave electric. By default, the value is '(t): makes it behave electric as soon as the erlang-electric-commands list includes erlang-electric-newline.
Electric < • ∑𝑥 Smartparens	**<**	(**erlang-electric-lt** &optional ARG)	Insert a less-than sign, and optionally mark it as an open paren. • When smartparens-mode is active **<<** automatically inserts the closing pair.
Electric > *Electric behaviour:* • *new line & indent*	**>**	(**erlang-electric-gt** &optional ARG)	Insert a greater-than sign, and optionally insert a new line and indent. • *Electric behaviour:* **->** force new line and indent. ☝ With PEL, you can also type **->** without electric behaviour by typing **-.** See below.
	M-1 >		*Disable electric behaviour for this character:* Just insert **>** by typing **M-1 >**
Insert -> by typing -.	**-.**	(**pel-erlang-electric-period** &optional arg)	Insert **->** when typing **-.** only if the following conditions are met (otherwise inserts **-.**) : • period is included in the pel-erlang-electric-keys user-option value • point is inside code and dash does not follow **$**, as in **$-**
Electric comma *Electric behaviour:* • *new line & indent* • *space after comma in block*	**,**	(**erlang-electric-comma** &optional ARG)	Insert a comma character and possibly: • a new indented line when the comma is at the end of an **Erlang expression**. • a space if inside a block and **pel-erlang-space-after-comma-in-block** user-option is on.
	M-1 ,		*Disable electric behaviour for this character:* Just insert **,** by typing **M-1 ,**
Electric semicolon *Electric behaviour:* • *insert clause function header*	**;**	(**erlang-electric-semicolon** &optional ARG)	Insert a semicolon character and possibly a **function clause head** *prototype* on the next line. • Behaves like the normal semicolon when supplied with a numerical arg, point is inside string or comment, or when there are non-whitespace characters following the point on the current line. • Inserts a function clause *head prototype* when the selection criteria identified by **erlang-electric-comma-criteria** indicates that it should be done.
			• **erlang-electric-semicolon-insert-blank-lines** sets # of lines inserted between the current line & new function header.
📦 smart-dash See: ∑ **Inserting Text**	• **_** • **<kp-subtract>**	(**smart-dash-insert**)	Insert underscore following [A-Za-z0-9_], dash otherwise. See: ∑ Inserting Text 📦 Requires **smart-dash** 🔄 activated by **pel-use-smart-dash**, or when smart-dash-mode is in **pel-erlang-activates-minor-modes**.
Filling Text See also: ∑ **Filling/Justification**	• Text wrapping and filling applies to all text in the Erlang buffer: code and comment. The auto-fill command will automatically wraps code and comments. • Filling Erlang code does not work as it treats code as normal text. But filling comment paragraphs is useful. • The **pel-erlang-fill-column** sets the **fill-column** variable to control where text wraps in Erlang buffers. • **pel-show-fill-column <f11> t f ?** shows its value. Use **set-fill-column** (**C-x f**) to set it. • Toggle display of a vertical line that shows it with **<f11> 8**.		
Fill current paragraph	• **M-q** • **<f11> t f p**	(**fill-paragraph** &optional JUSTIFY REGION)	Fill multi-line comment at or after point. • To justify as well: **C-u M-q** • In auto fill mode the text filling is done at the end of the line.

Description	Keystroke	Function	Note
Erlang Comments • Erlang Programming Rules & Conventions See also: ∑ Comments	Erlang uses the **%** character to identify line comments. It uses the following conventions: • **%** - Single percent characters for comments located toward the end of a line of code • **%%** - Two percent characters are used for comments starting at indentation level. • **%%%** - Three percent characters are used to describe modules and are always placed in the first column The location of the comment inside a code line is controlled by the **comment-column** variable. Set it with **comment-column,** bound to **C-x ;**		
Comment/un-comment • PEL extension of comment-dwim specialized for Erlang. Automatically uses the %%% comment when appropriate. ★★ **Note:** • **M-;** works much better than **C-c C-c** and **C-c C-u** • PEL maps **M-;** to **pel-erlang-comment-dwim** which works even better. See also: ∑ Comments	**M-;**	(**comment-dwim** ARG)	Comment line or region with **%** or **%%** style comments depending on the location in the buffer.
		(**pel-erlang-comment-dwim** &optional ARG)	Does the same but adds ability to insert **%%%** comments. It does that on the very first line in the buffer and lines that follow a line that starts with **%%%**.
	• When no marked region and no comment: On empty line: insert **%%** comment starter at the proper indentation level. On first empty line in buffer: insert **%%%** comment. Also following lines or region that starts with **%%%** On line with code: insert **%** comment starter after the code for an end-of-line comment • With marked un-commented region: Comment region (each line is commented) • With marked commented region: Un-comments the region. • To force insert **%%%** comment style: type **M-3 M-;**. The **M-3** prefix identifies 3 **%** characters to insert. You can use another number. ✂ The erlang.el code binds **M-1** to indent-for-comment. However PEL uses **M-1** for something else.		
	C-c C-c	(**comment-region** BEG END &optional ARG)	Comment or uncomment each line in the region. • With just **C-u** prefix arg, uncomment each line in region BEG .. END. • Numeric prefix ARG means use ARG comment characters. • If ARG is negative, delete that many comment characters instead.
	• The comment start is identified by '**comment-start**' and '**comment-padding**'; the comment end by '**comment-end**' and '**comment-padding**'. • By default, the '**comment-start**' markers are inserted at the current indentation of the region, and comments are terminated on each line (even for syntaxes in which newline does not end the comment and blank lines do not get comments). This can be changed with '**comment-style**'.		
Un-comment region	**C-c C-u**	(**uncomment-region** BEG END &optional ARG)	Uncomment each line in the BEG .. END region. The numeric prefix ARG can specify a number of chars to remove from the comment delimiters.
Toggle display of comments in buffer or active region	**<f11> ; ;**	(**hide/show-comments-toggle** &optional START END)	Toggle hiding/showing of comments in the active region or whole buffer. • If the region is active, then toggle comments in the region. Otherwise, in the whole buffer. 📦 Requires the **hide-comnt.el** package PEL activates it with **pel-use-hide-comnt**
Hard Tabs Rendering See also: ∑ Indentation **Hard Tab Display Rendering**	Like most programming languages, you can use hard tabs and spaces as horizontal whitespace in the Erlang source code. • Emacs supports all variations of styles: spaces only and mix of hard-tabs and spaces. Using only hard-tabs in Erlang is possible but rare. Some people use hard-tabs for indentation and extra spaces for alignment. Emacs supports all of these styles. • Emacs provides commands to convert code to remove all hard-tabs (**untabify**) and replace as many spaces as possible with hard tabs (**tabify**). • The **tab-width** user-option controls the *visual rendering* of hard tabs **not** the indentation level. • PEL provides an Erlang specific user option for hard-tab: **pel-erlang-tab-width** user-option. • PEL also provides the following command to dynamically modify the tab width rendering in the current buffer.		
Set visual rendering of hard tabs for the current buffer See ∑ Indentation	**<f11> M-t**	(**pel-set-tab-width** N)	Change the tab width of the current buffer, only affecting the display rendering of hard tabs inserted in the buffer text. Prompts for a new value in the [2, 8] range. • This modifies a buffer local value of the the **tab-width** user-option. • The change is temporary and affects the current buffer only. • To change the tab width used for all Erlang source code files, change the '**pel-erlang-tab-width**' user-option variable instead.
Hard Tab Insertion	The **pel-erlang-use-tabs** user-option controls whether hard tab characters are inserted in Erlang source code when Emacs inserts indentation whitespace. • This sets the Emacs **indent-tabs-mode** for Erlang buffers.		
Indentation **indentation**	All syntactic indentation control for Erlang is controlled by the erlang-mode logic and several user-options in the erlang group. See **indentation**. • Rigid indentation commands are also available and listed at the end of this list. They are also listed in the ∑ Indentation table.		
Indent current line or region See also: ∑ Indentation *Erlang Guidelines*: • **Ericsson AB**: try to limit most code to 2 levels of indentation. • **Inaka**: indentation level = 2 space chars.	**<tab>**	(**indent-for-tab-command** &optional ARG)	Indent active region, current line, or block starting on this line: performs syntactic indentation. • The indentation level is controlled by the **erlang-indent-level** user-option. Its default is 4. • Access its custom group buffer using **<f12> <f3> 1**
	• In Transient Mark mode, when the region is active, reindent the region. • Otherwise, with a prefix argument, rigidly reindent the expression starting on the current line. • Otherwise reindent just the current line. ☝ You can type **<tab>** ***anywhere*** in the line to indent the current line or everything in the marked area if a block is marked. • Note that the erlang.el logic doubles the indentation label inside funs. See this **S.O. discussion on that**. ☝ To ***indent rigidly*** you can use: • (**pel-indent-rigidly &optional N**) (bound to **C-x <tab>** and to **<f11> <tab><tab>**) to indent the line or region rigidly. • (**tab-to-tab-stop**), bound to **M-i** to insert spaces to the next tab stop column.		
Indent complete buffer	**<f12> <tab>**	(**erlang-indent-current-buffer**)	Indent current buffer as Erlang code. • Works on the entire buffer, even if it is narrowed.
Indent Erlang function	**C-c C-q** **<f12> f <tab>**	(**erlang-indent-function**)	Indent current Erlang function. Point can be located anywhere inside the function.
Indent function clause	**<f12> c <tab>**	(**erlang-indent-clause**)	Indent current Erlang clause. Point can be located anywhere in the Erlang clause.
Indent lines of list after point	**C-M-q**	(**prog-indent-sexp** &optional DEFUN)	Indent the expression after point. See also: ∑ Indentation When interactively called with prefix, indent the enclosing function instead.
Indent a region	**C-M-**	(**indent-region** START END &optional COLUMN)	Indent each nonblank line in the region. • A numeric prefix argument specifies a column: indent each line to that column.
	• With no prefix argument, the command chooses one of these methods and indents all the lines with it: 1. If 'fill-prefix' is non-nil, insert 'fill-prefix' at the beginning of each line in the region that does not already begin with it. 2. If 'indent-region-function' is non-nil, call that function to indent the region. 3. Indent each line via 'indent-according-to-mode'. ☝ When a region is marked you can also use the simple **<tab>** to do the same when syntactic-indentation is active.		
Outline Erlang Code See ∑ Outline for all key bindings	By activating the **outline-minor-mode** you can easily turn the Erlang buffer into an outline of function definitions. **outline-regex** & **outline-level** Once the minor mode is active you can collapse and expand code as outlines and navigate using the outline commands. See the key bindings in ∑ Outline ☝ This is very useful to quickly see an outline of the code in a large file. Using the outline-hide-other is particularly effective. • PEL binds the outline commands under the **<f2>** key prefix when the outline-minor-mode is active. Two useful key bindings are shown below.		
Toggle outline minor mode	**<f11> M-l**	(**outline-minor-mode** &optional ARG)	Toggle Outline minor mode. • Enable with a prefix positive argument ARG, disable with negative argument.
Hide other	• **<f2> o**	(**outline-hide-other**)	Hide everything except current body and parent and top-level headings. • This also unhides the top heading-less body, if any.
Show all	• **<f2> a**	(**outline-show-all**)	Show all of the text in the buffer.

Description	Keystroke	Function	Note
Navigation in Erlang code See also: • ⟫ Navigation • Moving by Defuns	The erlang-mode provides commands to navigate across Erlang source code. PEL complements these. And EDTS also. Several commands are specialization of the normal navigation commands which are described in the table ⟫ Navigation, but several are specific to Erlang: • Notice the 3 sets of commands: 1. **<f12> <up>** and **<f12> <down>** move to the beginning of Erlang functions skipping all compiler directives. 2. The standard navigation commands, (mapped to **<f6>** prefix) move to beginning/end of Erlang functions but stop at compiler directives. 3. The **<f12> <M-*cursor*>** commands (also accessible via **<M-f12> <M-*cursor*>**, move across Erlang clauses (as opposed to functions). The list below describe the specialized commands only. See the others inside ⟫ Navigation, like the navigation by blocks. ☞Note that all **<f12>** prefixes shown below are available in erlang-mode. Their global equivalent is **<f11> SPC e** . It is not always shown for brevity. ☞Some navigation examples use icons to represent point position. The start position is shown as 0 with following positions as 1 to 10.		
• **By Function**	• Move to next/previous function beginning/end at/skipping compiler directives. **Skips clauses**.		
• **to start of function**	• Move to beginning of function		
• **Go backward to beginning of previous function**	• **<f12> <up>** • **<f12> f p**	(**pel-previous-erl-function** &optional N)	Move backward to the beginning of the previous function skipping all compiler directives. • Moves point to the first character of the function name. • With prefix argument N repeat N times. • Pushes mark; move back to previous position with **M-`**. ☛Shift marking is available for the key sequence using a cursor key.
	• **<f11> SPC e <up>** • **<f11> SPC e f p**		
	C-c C-d C-b	(**ferl-goto-previous-function**)	Move backward to the beginning of the previous function. • Skips all compiler directives. 📦 Requires **EDTS** PEL activates it with **pel-use-edts** (set to **t** or **start-automatically**).
• **Go forward to beginning of next function**	• **<f12> <down>** • **<f12> f n**	(**pel-next-erl-function** &optional N)	Move forward to the beginning of the next function skipping all compiler directives. • Moves point to the first character of the function name. • With prefix argument N repeat N times. • Pushes mark; move back to previous position with **M-`**. ☛Shift marking is available for the key sequence using a cursor key.
	• **<f11> SPC e <down>** • **<f11> SPC e f n**		
	C-c C-d C-f	(**ferl-goto-next-function**)	Move forward to the beginning of the next function. • Skips all compiler directives. 📦 Requires **EDTS** PEL activates it with **pel-use-edts** (set to **t** or **start-automatically**).
• **to start of function/ directive**	• Move to beginning of function or compiler directive		
• **Go backward to beginning of previous:** • **function** • **compiler directive**	**<f12> f P**	(**beginning-of-defun** &optional ARG) (**erlang-beginning-of-function** &optional ARG)	Move backward to the beginning of an Erlang function or compiler directive. • 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>**. 🐞Erlang.el man page indicates an invalid mapping for this.
	• **C-M-a** • **C-M-<home>** • **<f6> <up>** • **<f11> SPC e f P**		
• **Go forward to beginning of next:** • **function** • **compiler directive**	**<f12> f N**	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next function definition or compiler directive. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`**. ☛Shift marking is available for the **<f6>** bindings : hold Shift after typing **<f6>**.
	• **<f6> <down>** • **<f11> SPC e f N**		
• **to end of function**	• Move to end of function or compiler directive		
• **Backward to end of previous:** • **function** • **compiler directive**	**<f6> <left>**	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to line after end of the previous function definition. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. • Move back to previous position with **M-`**. ☛Shift marking is available for the **<f6>** bindings.
• **Forward to end of next:** • **function** • **compiler directive**	• **C-M-e** • **C-M-<end>** • **<f6> <right>**	(**end-of-defun** &optional ARG) (**erlang-end-of-function** &optional ARG)	Move forward to line after end of Erlang function. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (for **C-M-e** and **C-M-<end>**). However **<f6> <right>** handle Shift-marking fine in terminal mode.
• **By Expression** • **functions, etc..**	Note that in Erlang every single expression or expression sequence ends with a period. Expressions in expression sequences are separated by commas. The following commands move to the beginning/end of single expression or expression sequence. • They do not move across expressions in a sequence of expressions. • Since Erlang function definition is also an Erlang expression, these commands move across function definitions.		
• **Go to beginning of statement**	**M-a**	(**backward-sentence** &optional ARG)	Go backward to the beginning of an Erlang statement. • With a numerical argument repeat that many times.
	<f12> s a		
• **Go to end of statement**	**M-e**	(**forward-sentence** &optional ARG)	Go forward to the end of an Erlang statement. • With a numerical argument repeat that many times.
	<f12> s e		
• **By Function Clause**	Move by clauses of a function. A function definition (statement) may have multiple clauses, each separated by a semicolon.		
• **Go backward to beginning of clause**	• **C-c M-a** • **<f12> c a** • **<M-f12> <M-up>**	(**erlang-beginning-of-clause** &optional ARG)	Move backward to previous start of clause. • With argument, do this that many times. 🐞Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.
• **Go forward to beginning of next clause**	• **<f12> c n** • **<M-f12> <M-down>**	(**pel-beginning-of-next-clause**)	Move forward to the beginning of next clause. • Pushes mark; move back to previous position with **M-`**. ☛Shift marking is available.
• **Go backward to end of previous clause**	• **<f12> c p** • **<M-f12> <M-left>**	(**pel-end-of-previous-clause**)	Move backward to the end of the previous clause. • Pushes mark; move back to previous position with **M-`**. ☛Shift marking is available.
• **Go forward to end of current clause**	• **C-c M-e** • **<f12> c e** • **<M-f12> <M-right>**	(**erlang-end-of-clause** &optional ARG)	Move to the end of the current clause. • With argument, do this that many times. 🐞Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.

Description	Keystroke	Function	Note
• **Block Navigation** See also: • ∑𝑥 Smartparens	Erlang syntax uses balanced blocks made out of the following character pairs, generically called *block parens*: • **( )** for function parameters, expression grouping • **{ }** for **tuples**, **records**, **maps** • **[ ]** for **lists** • **" "** for **strings** • **<< >>** for **binaries and bitstrings** • The smartparens-mode can be activated automatically for Erlang by adding erlang-mode to the **pel-erlang-activates-minor-modes** user-option. • Use the **<f11> ((** key sequence to toggle the smartparens-mode on and off. Standard Erlang support provide some commands to navigate across and into these balanced blocks. Their name is shown in black in the following rows. • Other commands are provided by ∑𝑥 Smartparens when smartparens-mode minor-mode is active. Some are PEL specializations of smartparens code.		
• **To Block start/end**	The following commands move to the beginning or end of a block, skipping over Erlang terms inside these blocks.		
• **Go backward to beginning of previous block** • **Skips terms.**	• **C-M-p**	(**backward-list** &optional ARG)	Move backward to beginning of previous block. • Supports blocks of (), [] and {}. • With ARG, do it that many times. • A **negative argument** N means forward-list N. • This command assumes point is not in a string or comment. -spec ejabberd_started() -> ok. ejabberd_started() -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded() -> ok. config_reloaded() -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT).
• **Go backward to end of previous block** • **Skips terms.** • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> p**	(**pel-sp-previous-sexp** &optional ARG)	Move backward to end of previous block. • With ARG, do it that many times. If there is no next expression at current level, jump one level up (effectively doing 'sp-up-sexp'). • A **negative argument** N means move to the end of N-th following balanced expression. -spec ejabberd_started() -> ok. ejabberd_started() -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded() -> ok. config_reloaded() -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT).
• **Go forward to end of next block** • **Skips terms.**	• **C-M-n**	(**forward-list** &optional ARG)	Move forward to end of next block. • Supports blocks of (), [] and {}. • With ARG, do it that many times. • A **negative argument** N means forward-list N. • This command assumes point is not in a string or comment. -spec ejabberd_started() -> ok. ejabberd_started() -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded() -> ok. config_reloaded() -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT).
• **Go forward to beginning of next block** • **Skips terms.** • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> n**	(**pel-sp-next-sexp** &optional ARG)	Move forward to **beginning** of next block (and term if 'sp-navigate-consider-symbols' is set). • With ARG, do it that many times. • If there is no next expression at current level, jump one level up (effectively doing 'sp-backward-up-sexp'). -spec ejabberd_started() -> ok. ejabberd_started() -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded() -> ok. config_reloaded() -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT).
• **By Blocks and Terms** See also: • ∑𝑥 Smartparens	Move across blocks made of pairs of **{}**, **[]** and **()**. Also stops at terms. ⚠ With PEL: to use **Esc C-<left>** and **Esc C-<right>** bindings below, set **pel-windmove-on-esc-cursor** user-option is set to nil. Several Linux distros map **C-M-<left>** and **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. ☝ PEL enhances behaviour of these keys by providing the ability to move across Erlang's << >> bit syntax statement blocks.		
• **Go backward to beginning of previous term/block**	• **C-M-<left>** • **C-[C-b** • **Esc C-b** • **Esc C-<left>** ⚠ • **C-M-b**	(**pel-erlang-backward-sexp** &optional ARG)	Move backward backward to beginning of previous term or block. • With ARG, do it that many times. • A negative arg N means move forward to end of N terms/blocks. • At beginning of block, jump out of the current one. • 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. • **C-M-b** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<left>** : ☛ Shift marking works with this command. ❖ **C-M-<left>** does not work on Windows, but **H-<left>** works.
• ∑𝑥 Smartparens **with smartparens-mode active:** • **C-M-b** and **<M-f7> b** use **sp-backward-sexp,** • others are using **backward-sexp**	• **C-M-b** • **<M-f7> b**	(**sp-backward-sexp** &optional ARG)	Same as above with the additional behaviour: • With 'sp-navigate-consider-symbols' symbols and strings are also considered balanced expressions. It is set by default. • When it is nil, point only stops at 1, 4, 6 and 9: it jumps over terms. -spec ejabberd_started() -> ok. ejabberd_started() -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded() -> ok. config_reloaded() -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT). Inside a block: gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT).

Description	Keystroke	Function	Note
• **Go forward to end of next term/block**	• **C-M-<right>** • **C-[C-f** • **Esc C-f** • **Esc C-<right>** ⚠️ • **C-M-f**	(**pel-erlang-forward-sexp** &optional ARG)	Move forward to end of term or block. • With ARG, do it that many times. • A **negative argument** N means move backward to beginning of previous term or block. • At end of block, jump out of the current one. • **C-M-n** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-f** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<right>** : ☛ Shift marking works with this command. ❖ **C-M-<right>** does not work on Windows, but **H-<right>** does.
• ∑𝑥 Smartparens **with smartparens-mode active:** • **C-M-f** and **<M-f7> f** use **sp-forward-sexp,** • others are using **forward-sexp**	• **C-M-f** • **<M-f7> f**	(**sp-forward-sexp** &optional ARG)	Same as above with the additional behaviour: • With 'sp-navigate-consider-symbols' symbols and strings are also considered balanced expressions. It is set by default. • When it is nil, point only stops at 3, 6 and 9: it jumps over terms. 0-spec1 ejabberd_started2()3 -> ok4. ejabberd_started5()6 -> gen_server7:call8(?MODULE, ejabberd_started, ?CALL_TIMEOUT)9. -spec10 config_reloaded() -> ok. config_reloaded() -> gen_server:call(0?MODULE1, config_reloaded2, ?CALL_TIMEOUT3).
• **Into block**	Navigate inside nested blocks of elements with the following commands.		
Into block forward	**C-M-d**	(**down-list** &optional ARG)	Move forward to the beginning of inner element of a block. • With ARG, do this that many times. • A **negative argument** N means move backward but still go down a level. • If ARG is raw prefix argument **C-u**, descend forward as much as possible. • If ARG is raw prefix argument **C-u C-u**, jump to the beginning of current list. • If the point is inside block and there is no down expression to descend to, jump to the beginning of current one. If moving backwards, jump to end of current one. music_info() -> 0{1{2error, {3noreply, State}}, ☛ example {good, {{year, 1974}, {group, "Contraction"}, 0[1{2song, "3Sam M'Madown"}, ☛ example {song, "A la claire fontaine"}, {song, "L'alarme à l'oeil"}, {song, "La bourse ou la vie"}] {rating, excellent}}}}.
• ∑𝑥 Smartparens **with smartparens-mode active**	• **C-M-d** • **<M-f7> d**	(**sp-down-sexp** &optional ARG)	
Into block backward • ∑𝑥 Smartparens **with smartparens-mode active**	• **<M-f7> z** • **C-M-z**	(**sp-backward-down-sexp** &optional ARG)	Move backward down one level to end of block element. • With ARG, do this that many times. • A **negative argument** N means move forward but still go down a level. • If ARG is raw prefix argument **C-u**, descend backward as much as possible. • If ARG is raw prefix argument **C-u C-u**, jump to the end of current list. • If the point is inside sexp and there is no down expression to descend to, jump to the end of current one. If moving forward, jump to beginning of current one. music_info(1) -> 0{{error, {noreply, State}}, ☛ example {good, {{year, 1974}, {group, "Contraction"}, [{song, "Sam M'Madown"}, {song, "A la claire fontaine"}, {song, "L'alarme à l'oeil"}, {song, "La bourse ou la vie"}] {rating, excellent4}3}2}1}.0 ☛ example
• **to edge of block**			
To beginning of block • ∑𝑥 Smartparens **with smartparens-mode active**	• **<M-f7> a**	(**sp-beginning-of-sexp** &optional ARG)	Jump to beginning of the block the point is in. • The beginning is the point after the opening delimiter. • With no argument, this is the same as **C-u C-u 'sp-down-sexp'** • With ARG positive **N > 1**, move forward out of the current expression, move N-2 expressions forward and move down one level into next expression. • With ARG negative **N < 1**, move backward out of the current expression, move N-1 expressions backward and move down one level into next expression. • With ARG raw prefix argument **C-u** move out of the current expressions and then to the beginning of enclosing expression. music_info() -> {{error, {noreply, State}}, {good, {{1year, 1904}, ☛ example {group, "1Contract0ion"}, ☛ example [1{song, "Sam M'Madown"}, {song, "A la claire fontaine"}, {song, "L'alarme à l'oeil"}, {song, "La bourse ou la vie"}0] ☛ example {rating, excellent}}}}.
To end of current block • **forward** • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> e**	(**sp-end-of-sexp** &optional ARG)	Jump to end of the current block. • With no argument, this is the same as calling **C-u C-u 'sp-backward-down-sexp'**. • With ARG positive **N > 1**, move forward out of the current expression, move N-1 expressions forward and move down backward one level into previous expression. • With ARG negative **N < 1**, move backward out of the current expression, move N-2 expressions backward and move down backward one level into previous expression. • With ARG raw prefix argument **C-u** move out of the current expressions and then to the end of enclosing expression. music_info() -> {0{error, {noreply, State}1}, ☛ example {0good, {{year, 1974}, ☛ example 👇 {group, "Contraction"}, [{song, "Sam M'Madown"}, {song, "A la claire fontaine"}, {song, "L'alarme à l'oeil"}, {song, "La bourse ou la vie"}] {rating, excellent}}1}}.

Description	Keystroke	Function	Note
• **Out of block**			
Out block forward • **forward** • ∑𝑥 Smartparens **with smartparens-mode active**	**C-M-]**	(**up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward out of one level of block parens. • With ARG, do this that many times. • A **negative argument** means move backward but still to a less deep spot. • If called interactively and 'sp-navigate-reindent-after-up' is enabled for current major-mode, remove the whitespace between end of the expression and the last "thing" inside the expression. • This behaviour can be suppressed for syntactic string blocks by setting 'sp-navigate-reindent-after-up-in-string' to nil. • If 'sp-navigate-close-if-unbalanced' is non-nil, close the unbalanced expressions automatically. •
	• **C-M-]** • **<M-f7>]**	(**sp-up-sexp** &optional ARG INTERACTIVE)	
Out block backward • **backward** • ∑𝑥 Smartparens **with smartparens-mode active**	• **<M-f7> u** • **C-M-u**	(**sp-backward-up-sexp** &optional ARG INTERACTIVE)	Move backward out of one level of block parens. • With ARG, do this that many times. • A **negative argument** means move forward but still to a less deep spot. • If called interactively and 'sp-navigate-reindent-after-up' is enabled for current major-mode, remove the whitespace between beginning of the expression and the first "thing" inside the expression.
Move over space	The commands all use the ∑𝑥 Smartparens external package and required smartparens-mode minor-mode to be active. 🚧 Current implementation of sp-forward-symbol and sp-backward-symbol stop inside comments. I consider this a bug 🐞 so I **reported** and submitted a potential **fix**. Until these are integrated PEL implement workaround commands that do not stop inside comments: **pel-sp-forward-symbol** and **pel-sp-backward-symbol**. PEL binds the key sequence to those until the fix is integrated.		
To beginning of next symbol/block • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> SPC n**	(**sp-skip-forward-to-symbol** &optional STOP-AT-STRING STOP-AFTER-STRING STOP-INSIDE-STRING)	Skip whitespace and comments moving forward. • If STOP-AT-STRING is non-nil, stop before entering a string (if not already in a string). • If STOP-AFTER-STRING is non-nil, stop after exiting a string. • If STOP-INSIDE-STRING is non-nil, stop before exiting a string.
To end of next symbol or block • ∑𝑥 Smartparens **with smartparens-mode active** See 🚧 🐞 note above.	**<M-f7> SPC m**	(**pel-sp-forward-symbol** &optional ARG)	Move point to the next position that is the end of a symbol. • With ARG being positive number N, repeat that many times. • With ARG being negative number -N, repeat that many times in backward direction. • A symbol is any sequence of characters that are in either the word constituent or symbol constituent syntax class. Current symbol only extend to the possible opening or closing delimiter as defined by 'sp-add-pair' even if part of this delimiter would match "symbol" syntax classes.
To beginning of previous • ∑𝑥 Smartparens **with smartparens-mode active** See 🚧 🐞 note above.	**<M-f7> SPC p**	(**pel-sp-backward-symbol** &optional ARG)	Move point to the next position that is the beginning of a symbol. • With ARG being positive number N, repeat that many times. • With ARG being negative number -N, repeat that many times in forward direction. • A symbol is any sequence of characters that are in either the word constituent or symbol constituent syntax class. Current symbol only extend to the possible opening or closing delimiter as defined by 'sp-add-pair' even if part of this delimiter would match "symbol" syntax classes.
Skip forward past whitespace • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> SPC .**	(**sp-forward-whitespace** &optional ARG)	Skip forward past the whitespace characters. • With non-nil ARG return number of characters skipped.
Skip backward past whitespace • ∑𝑥 Smartparens **with smartparens-mode active**	**<M-f7> SPC ,**	(**sp-backward-whitespace** &optional ARG)	Skip backward past the whitespace characters. • With non-nil ARG return number of characters skipped.

Description	Keystroke	Function	Note
Cross Reference navigation See ∑ Xref **See PEL Manual Erlang Cross Reference section for comparison of available methods.**	Erlang cross reference navigation, that uses the **M-.** key to move to the definition of the thing at point, is supported by several tools: • The xref-based cross reference tools with the following backends: • etags (with etags or CTags generated tags file), use **etags-erl** shell script to create a TAGS file in the directory root to use with etags. • 📦 Global/gtags with **ggtags**. Source the **envfor-gtags** shell script to set up your shell before starting Emacs to use gtags. • You must install 🧰 **GNU Global** for this. See **PEL manual installation instructions for GNU Global**. • With PEL set **pel-use-ggtags** user-option to t to install the Emacs-side support **ggtags** package and activate the gtags commands. • 📦 **dumb-jump** to navigate without having to create external database or tags files. Set **pel-use-dumb-jump** on to activate it. • For the above use the **<f11> X <f2>** key sequence to access PEL customization buffer for cross reference control. • Other specialized tools for Erlang: • 📦 **ivy-erlang-complete** external package activated by **pel-use-ivy-erlang-complete** user-option. • 🧰 Requires a version of Erlang installed that supports **Erlang escript**. ⚠️ ivy-erlang-complete replies on GNU sed, which is not accessible on macOS by default. • Install gnu-sed with Homebrew. I provided a patch which solves the problem by detecting macOS and using **gsed** instead of **sed**. • 📦 The **EDTS** external package. activated by **pel-use-edts** user option. • 📦 The **lsp-mode** external package activated by the **pel-use-erlang-ls** user-option. Both **EDTS** and **lsp-mode** use launch an Erlang node or server to operate. None of the other tools do. **ivy-erlang-complete** parse Erlang code and is project and Erlang library aware providing a good user experience without having to launch an Erlang node or server. **dumb-jump** uses fast search tools but has limited Erlang knowledge. The other Xref-based tools require a TAGS (etags) or a database (**ggtags**) that must be setup prior to use. The **ggtags** tool provide more capabilities than etags, which is built in Emacs and does not require any other external package.		
PEL Unified Cross Reference Navigation	Each of the above tools use different back-end function. Some of these are accessible via Emacs unified Xref mechanism but not all. • PEL unifies all of these tools allowing you to select the one you prefer to use via customization and also allowing you to change the tool during and editing session. 🎛 Select the default cross reference engine by setting the **pel-erlang-xref-engine** user-option. 🗡 Modify the cost reference engine during an editing session with **<M-f12> M-. M-.** . Display which one is used with **<M-f12> M-. M.?** To move point to the definition of an identifier, place point over it and type the usual **M-.** key. It will use the currently selected cross reference engine.		
🗡 **Select Cross Reference back-end for Erlang**	**<M-f12> M-. M-.**	(**pel-erlang-select-xref**)	Select another Erlang cross reference back-end from the back-ends currently available. • The selection remains active for the current editing session. • The '**pel-erlang-xref-engine**' user-option identifies the persistent selection.
Show selected Erlang Cross Reference back-ends	**<M-f12> M-. M-?**	(**pel-erlang-show-xref**)	Show Erlang cross reference back-end selected by customization and the one currently active.
Find definition of identifier at point using currently active engine ★★★ See also: ∑ Xref	**M-.**	(**pel-erlang-find-definitions**)	Grab symbol at point and move cursor to its definition. • Uses the currently active Erlang cross-reference back-end selected by '**pel-erlang-xref-engine**' user-option or modified via **<M-f12> M-. M-.** • If there are more than one match, prompt in the *xref* buffer. • For the Xref-driven back-ends: to search for a symbol entered manually, type **C-u M-.**
Go back to where M-. was last issued	**M-,**	(**xref-pop-marker-stack**)	• Pop back to where M-. was last invoked. • Marker depth is controlled by the **xref-marker-ring-length** user option.
EDTS/Cross References	**EDTS** provides the following cross-reference commands. It supports navigating in Erlang source code running in the current and remote nodes. PEL unbinds EDTS **M-.** and **M-,** to allow EDTS to work with PEL unified cross reference mechanism, and creates the bindings under **C-c C-d**. 📦 Requires the **EDTS** external package. activated by **pel-use-edts** user option. ☝ PEL integrates **EDTS** cross reference navigation in the unified cross reference navigation. • While EDTS is active you can use EDTS cross reference mechanism or anything selectable by **pel-erlang-select-xref** as described above. • If another cross reference engine is active and **EDTS** is on, you can force using the **EDTS** commands using the **C-c C-d** key bindings shown below.		
EDTS **Find definition of identifier at point**	~~M-.~~ **C-c C-d M-.**	(**edts-find-source-under-point**)	Goto the source code that: defines the function being called at point or header file included at point. For remote calls, contacts an Erlang node to determine which file to look in, with the following algorithm: • Find the directory of the module's beam file (loading it if necessary). • Look for the source file in: • Directory where source file was originally compiled. • Todo: Same directory as the beam file • Todo: Again with /ebin/ replaced with /src/ • Todo: Again with /ebin/ replaced with /erl/ Otherwise, report that the file can't be found.
EDTS: Go back to where M-. was last issued	~~M-,~~ **C-c C-d M-,**	(**edts-find-source-unwind**)	Unwind back from uses of 'edts-navigate'-commands.
Lists caller of function at point	• **C-c C-d w** • **<f12> w**	(**edts-xref-who-calls**)	Pops-up a menu of all callers of the function at point.
List the callers again	• **C-c C-d W** • **<f12> W**	(**edts-xref-last-who-calls**)	Redo previous call to edts-who-calls.
Find a function in the current module	• **C-c C-d f** • **<M-f12> M-f**	(**edts-find-local-function** SET-MARK)	Find a function in the current module. • List local functions in the mini-buffer. Support completion. Move point to selected one. • With **C-u** prefix, push mark before moving point.
Find a module in the current project	• **C-c C-d F** • **<M-f12> M-g**	(**edts-find-global-function**)	Find a module in the current project. • List project modules in the mini-buffer. Support completion. Open the file of selected one.
ivy-erlang-complete Cross References	**ivy-erlang-complete** provides the following functions to navigate across Erlang code. 📦 Requires **ivy-erlang-complete** external package activated by **pel-use-ivy-erlang-complete** user-option. ☝ PEL integrates **ivy-erlang-complete** cross reference navigation in the unified cross reference navigation. • You can use its cross reference mechanism or anything selectable by **pel-erlang-select-xref** as described above. • While another cross reference engine is active you use **ivy-erlang-complete** by using the key bindings under **C-c** as shown below.		
Find definition of identifier at point	• **C-c M-.** • ~~M-.~~	(**ivy-erlang-complete-find-definition**)	Find Erlang definition using ivy-erlang-complete.
	• **C-c M-?** • ~~M-?~~	(**ivy-erlang-complete-find-references**)	Find erlang references. • Use **M-,** to go back to original location.
	• **<f12> M-f** • **C-c C-f**	(**ivy-erlang-complete-find-spec**)	Find spec at point, with ivy completion listing all found, then opening source file. • It also find callback definition.
Open file at point.	• **<f12> M-o** • **C-c C-o**	(**ivy-erlang-complete-find-file**)	Open file at point. Find file in current project.

Description	Keystroke	Function	Note
Open file or web-page whose name is at point ★★ See also: • ∑ File-mngt • ∑ Key-Chords • ∑ Completion/Input	• C-^ • <f11> f . • <M-f11> M-f M-. • 6y	(**pel-open-at-point** &optional N)	Open the file, library or the URL, named at point, with potential line & column #s. The **6y** key-chord is available if **pel-use-key-chord** is non-nil. Command prefixes are supported with the key-chord. See ∑ **Key-Chords**. • Key prefix controls the window into which the file is open. • It's also possible to open the file inside the OS default web browser or application associated with the file type by using the **M-9** command prefix. • See ∑ **File-mngt** description of this command and the function docstring.
	This command is able to find source files in Erlang root directory and project tree, including inside the Erlang project dependencies (normally stored inside the deps directory tree created when building the project. The following user option controls this behaviour: • The **pel-erlang-project-root-identifiers** user-option identifies the files that are used as markers of Erlang project directory root. • The **pel-project-root-identifiers** user-option identifies the files used to identify the project root in general. • This includes the file .pel-project you can use if nothing in the list works for you. The search for file supports glob characters and partial directory path. • For example issuing the command over the string "something.?hl" in the source code will find all files named something.erl and something.hrl inside Erlang root directory tree and the current project. • The command prompts when several files are found. • It uses the prompting method selected by **pel-prompt-read-method** user-option. The default is a very primitive function implemented by PEL. You can select a more powerful **ivy** prompting instead. If you select that PEL will automatically set **pel-use-ivy** to **t** and Ivy mode will be installed automatically when you restart Emacs.		
Completion	Completion is available from various sources. • Without help from EDTS or LSP, the ivy-erlang-complete external package parses the Erlang libraries to identify the supported functions. • ivy-erlang-complete external package activated by **pel-use-ivy-erlang-complete** user-option. • Requires a version of Erlang installed that supports **Erlang escript**. ivy-erlang-complete replies on GNU sed, which is not accessible on macOS by default. • Install gnu-sed with Homebrew. I provided a patch which solves the problem by detecting macOS and using **gsed** instead of **sed**.		
Hippie Expand Abbreviation See also: ∑ **Hide/Show**	M-/	(**hippie-expand** ARG)	Try to expand text before point, using multiple methods. • Not an Erlang completion command but it can be useful to pick up names present in the files. • The expansion functions in '***hippie-expand-try-functions-list***' are tried in order, until a possible expansion is found. Repeated application of 'hippie-expand' inserts successively possible expansions. • With a positive numeric argument, jumps directly to the ARG next function in this list. With a negative argument or just **C-u**, undoes the expansion. PEL activates this when the **pel-use-hippie-expand** user option is set to **t**.
Completion of Erlang code at point.	<f12> . C-:	(**ivy-erlang-complete**)	Erlang completion at point. • Aware of Erlang modules and functions for the currently used Erlang version identified by the ivy-erlang-complete-erlang-root user-option which is adjusted to the erlang-root-dir ivy-erlang-complete replies on GNU sed, which is not accessible on macOS by default. • To solve the problem you must **install gnu-sed with Homebrew** since ivy-erlang-complete shell scripts use gsed instead of sed.
Open web-based Erlang standard library documentation for function at point	• <f12> M-h • C-c C-h	(**ivy-erlang-complete-show-doc-at-point**)	Show web-based Erlang standard library documentation for function at point.
Set a different root for Erlang project	• <f12> M-e • C-c C-e	(**ivy-erlang-complete-set-project-root**)	Set root for current project for ivy-erlang-complete. To see the current value of the ivy-erlang-complete-project-root, type <f12> ?
Marking See also: ∑ **Marking**	These commands complement what is already available and described in the ∑ **Marking** table. • The first 2 command listed below are Erlang-mode specific marking functions. • For those 2 commands the Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314. • The useful **er/expand-region** benefits from PEL enhancement to erlang syntax table supporting the < > pair therefore it is also mentioned here.		
Mark Erlang function	• C-M-h	(**mark-defun** &optional ARG)	Put mark at end of this function, point at beginning. • The function marked is the one that contains point or follows point. • With positive ARG, mark this and that many next functions; with negative ARG, change the direction of marking. • If the mark is active, it marks the next or previous function(s) after the one(s) already marked.
	• <f12> f m	(**erlang-mark-function** &optional ARG)	
Mark Erlang Clause	• C-c M-h • <f12> c m	(**erlang-mark-clause**)	Put mark at end of clause, point at beginning.
Mark region by semantic unit, increase marked region on each invocation. **★ ★ Powerful command** **Works best with superword-mode on**. • See ∑ **Text Modes**	• M-=	(**er/expand-region** ARG)	Increase selected region by semantic units. Requires the **expand-region** package, activated by ***pel-use-expand-region*** user option.
	• <f11> . =		
	• With prefix argument expands the region that many times. • If prefix argument is negative calls 'er/contract-region'. • If prefix argument is 0 it resets point and mark to their state before calling 'er/expand-region' for the first time. This command is very powerful: the first time it's typed it selects a word, if you type it again it will expand the selection, and again, and again. The expansions follow the semantics of the current major mode: it uses syntactic information from the major mode. ☛ 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 also 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.		

Description	Keystroke	Function	Note
Copy and Clone • ∑𝑥 Smartparens	The following commands provides specialized copy and cloning operations. They are provided by ∑𝑥 Smartparens • With PEL the commands that are marked with 👁 display the copied string when **pel-show-copy-cut-text** is **t**. Toggle this display with **<f11> M-=**		
Copy current & forward block(s) 👁	**<M-f7> =**	(**sp-copy-sexp** &optional ARG)	Copy the following ARG expressions to the kill-ring. This is exactly like calling 'sp-kill-sexp' with second argument t. All the special prefix arguments work the same way.
Copy previous block(s) 👁	**<M-f7> M-=**	(**sp-backward-copy-sexp** &optional ARG)	Copy the previous ARG expressions to the kill-ring. This is exactly like calling 'sp-backward-kill-sexp' with second argument t. All the special prefix arguments work the same way.
clone current block	**<M-f7> c**	(**sp-clone-sexp**)	Clone sexp after or around point. • If the form immediately after point is a sexp, clone it below the current one and put the point in front of it. • Otherwise get the enclosing sexp and clone it below the current enclosing sexp.
Transform code	The following commands can be used to help transform code. Some need external packages.		
iEdit mode 🎛 See also: ∑ Highlight	iEdit Mode - **Edit multiple instances of variable/symbols simultaneously.** ☝ This mode is **very useful** to rename symbols or variable during refactoring. 📦 Requires the **iedit** external package. PEL activates it with **pel-use-iedit**.		
Toggle iedit mode See also: • ∑ Cursor • ∑ Search/Replace	• **C-;** • **<f11> e** • **<f11> h i** • **<f11> m i**	(**iedit-mode** &optional ARG)	Toggle iEdit mode: **edit all symbols in scope or region simultaneously**. ⚠ Both iEdit and Flyspell use the **C-;** key as their default binding. • PEL detects and reports that situation: modify the binding of one of them if you see it. ➤ See ∑ Search/Replace where all the iedit-mode commands are described.
Align arrows inside region	**C-c C-a**	(**erlang-align-arrows** START END)	Align arrows ("->") in **function clauses** inside marked region or in the current function. • **With a prefix argument**, aligns **all arrows** *in the region* (or from beginning of buffer up to point), not just those in function clauses. **Before:** `sum(L)  -> sum(L, 0).` `sum([H|T], Sum) -> sum(T, Sum + H);` `sum([], Sum) -> Sum.` **After C-c C-a:** `sum(L)  -> sum(L, 0).` `sum([H|T], Sum) -> sum(T, Sum + H);` `sum([], Sum)  -> Sum.` To align something else than clauses, select the code and type: **C-u C-c C-a** **Before:** `check(P, [H|T]) ->` `  case P(H) of` `  true -> 1;` `  false -> 0` `  end;` **After C-u C-c C-a:** `check(P, [H|T]) ->` `  case P(H) of` `  true -> 1;` `  false -> 0` `  end;`
Transpose block elements • ∑𝑥 Smartparens with smartparens-mode active 🎛	**<M-f7> t**	(**sp-transpose-sexp** &optional ARG)	Transpose the expressions around point. • The operation will move the point after the transposed block, so the next transpose will "drag" it forward. • With arg positive N, apply that many times, dragging the expression forward. • With arg negative -N, apply N times backward, pushing the word before cursor backward. This will therefore not transpose the expressions before and after point, but push the expression before point over the one before it. **Before** (for all following examples): `AList = [1, 2, 3,|[10,11,12,[22,33,44]], 5, 6, 7, 8,[]].` **After <M-f7> t**: `AList = [1, 2, [10,11,12,[22,33,44]], 3|, 5, 6, 7, 8,[]].` **After M-2 <M-f7> t**: `AList = [1, 2, [10,11,12,[22,33,44]], 5, 3|, 6, 7, 8,[]].` **Before** (for all following examples): `AList = [{first,[1, 2, 3]}|, [10,11,12,[22,33,44]], 5, 6, 7, 8,[]].` **After <M-f7> t**: `AList = [[10,11,12,[22,33,44]], {first,[1, 2, 3]}|, 5, 6, 7, 8,[]].` **After M-2 <M-f7> t**: `AList = [[10,11,12,[22,33,44]], 5, {first,[1, 2, 3]}|, 6, 7, 8,[]].` **Before** (for all following examples): `AList = [{first,[1, 2, 3]}|, [10,11,12,[22,33,44]], 5, 6, 7, 8,[]].` **After M- - <M-f7> t**: `AList = [{first,[1, 3|, 2]}, [10,11,12,[22,33,44]], 5, 6, 7, 8,[]].`
Push current block after next • ∑𝑥 Smartparens with smartparens-mode active 🎛	**<M-f7> s**	(**sp-push-hybrid-sexp**)	Push the hybrid sexp after point over the following one. **Before**: `AList = [1, 2, 3,` `  |[10,11,12,[22,33,44]],` `  5, 6, 7, 8,[]].` **After <M-f7> s**: `AList = [1, 2, 3,` `  |5, 6, 7, 8,[],` `  [10,11,12,[22,33,44]]].`
Transform - barf	The following commands extract members from block		
Eject next element(s) out of current block • ∑𝑥 Smartparens with smartparens-mode active 🎛	**<M-f7> /**	(**sp-forward-barf-sexp** &optional ARG)	Remove the last sexp in the current list by moving the closing delimiter. • If ARG is positive number N, barf that many expressions. • If ARG is negative number -N, contract the opening pair instead. • If ARG is raw prefix C-u, barf all expressions from the one after point to the end of current list and place the point before the closing delimiter of the list. • If the current list is empty, do nothing. 💻 smartparens by itself fails to process these examples properly. PEL fixes the issues with post processing. **Before:** `AList = [[1, |2, 3, 4]].` **After <M-f7> /**: `AList = [[1, |2, 3], 4].` **Before:** `AList = [[1, |2, 3, 4]].` **After M-2 <M-f7> /**: `AList = [[1, |2], 3, 4].` **Before:** `AList = [[1, |2, 3, 4]].` **After M- - <M-f7> /**: `AList = [1, |[2, 3, 4]].`
Eject previous element(s) out of current block • ∑𝑥 Smartparens with smartparens-mode active 🎛	**<M-f7> M-/**	(**sp-backward-barf-sexp** &optional ARG)	This is exactly like calling 'sp-forward-barf-sexp' with minus ARG. • In other words, instead of contracting the closing pair, the opening pair is contracted. For more information, see the documentation of 'sp-forward-barf-sexp'. This command works fine in Erlang for the following code examples: **Before:** `AList = [[1, |2, 3, 4]].` **After <M-f7> M-/**: `AList = [1, |[2, 3, 4]].` **Before:** `AList = [[1, |2, 3, 4]].` **After M-3 <M-f7> /**: `AList = [1, |2, 3, [4]].`

Description	Keystroke	Function	Note
Transform - slurp	The following commands perform slurping operations, however support for Erlang could be improved as the commands do not always work properly.		
Enclose next outside element into current block • ∑𝔵 Smartparens 	**<M-f7> >**	(**sp-forward-slurp-sexp** &optional ARG)	Add sexp following the current list in it by moving the closing delimiter. • If the current list is the last in a parent list, extend that list (and possibly apply recursively until we can extend a list or end of file). • If ARG is N, apply this function that many times. • If ARG is negative -N, extend the opening pair instead (that is, backward). • If ARG is raw prefix C-u, extend all the way to the end of the parent list. • If both the current expression and the expression to be slurped are strings, they are joined together. ⚠ This command does not always work well for Erlang as shown in the first example. • Use the next command for Erlang in those cases.
		💻 smartparens by itself fails to process these examples properly. PEL fixes the behaviour by using ability to post-process code to ensure correct syntax.	**Before:** `Names = [|]Joe.` **After <M-f7> >:** `Names = [|Joe].` **Before:** `AList = [[1, 2, 3 |], 4, 5].` **After <M-f7> >:** `AList = [[1, 2, |3, 4], 5].` **Before:** `AList = [1, 2, 3,` `[|10,11,12,[22,33,44]],` `5, 6, 7, 8,[]].` **After M-- <M-f7> >:** `AList = [1, 2, [3,` `| 10,11,12,[22,33,44]],` `5, 6, 7, 8,[]].`
Enclose previous outside element(s) into next block • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> <**	(**sp-backward-slurp-sexp** &optional ARG)	Add the sexp preceding the current list in it by moving the opening delimiter. • If the current list is the first in a parent list, extend that list (and possibly apply recursively until we can extend a list or beginning of file). • If arg is N, apply this function that many times. • If arg is negative -N, extend the closing pair instead (that is, forward). • If ARG is raw prefix C-u, extend all the way to the beginning of the parent list. • If both the current expression and the expression to be slurped are strings, they are joined together.
		The position of point inside the list does not matter. The point does not move.	**Before:** `AList = [0, 1, [2, 3|, 4], 5].` **After <M-f7> <:** `AList = [0, [1, 2, 3, 4], 5].` **Before:** `AList = [0, 1, [2, 3|, 4], 5].` **After M-2 <M-f7> <:** `AList = [[0, 1, 2, 3|, 4], 5].` **Before:** `AList = [-2, -1, 0, 1, [2, 3, 4|], 5].` **After C-u <M-f7> <:** `AList = [[-2, -1, 0, 1, 2, 3, 4|], 5].`
Enclose next element(s) into previous block • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> }**	(**pel-sp-add-to-previous-sexp** &optional ARG)	Add the expression around point to the first list preceding point. • With ARG positive N add that many expressions to the preceding list. • If ARG is raw prefix argument C-u add all expressions until the end of enclosing list to the previous list. • If ARG is raw prefix argument C-u C-u add the current list into the previous
		💻 smartparens by itself fails to process these examples properly. PEL fixes the issues with post processing and wrapping function.	**Before:** `AList = [0, 1, [2, 3], |4, 5].` **After <M-f7> }:** `AList = [0, 1, [2, |3, 4], 5].` **Before:** `AList = [0, 1, [2, 3], |4, 5].` **After M-2 <M-f7> }:** `AList = [0, 1, [2, |3, 4, 5]].`
Enclose previous outside element(s) into next block • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> {**	(**sp-add-to-next-sexp** &optional ARG)	Add the expressions around point to the first list following point. • With ARG positive N add that many expressions to the following list. • If ARG is raw prefix argument C-u add all expressions until the beginning of enclosing list to the following list. • If ARG is raw prefix argument C-u C-u add the current list into the following list.
		This command works fine in Erlang for the following code examples:	**Before:** `AList = [1, |2, [3, 4]].` **After <M-f7> {:** `AList = [1, |[2, 3, 4]].` **Before:** `AList = [1, |2, [3, 4]].` **After C-u <M-f7> {:** `AList = [[1, |2, 3, 4]].` **Before:** `AList = [[1, |2], [3, 4]].` **After C-u C-u <M-f7> {:** `AList = [[[1, |2], 3, 4]].`
Re-wrap block	Use the following commands to change the wrapping character pair surrounding a block		
Re-wrap current block • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> r**	(**sp-rewrap-sexp** PAIR &optional KEEP-OLD)	Re-wrap current block using another block character. Prompt for the pair beginning character. • With **C-u**, keep old delimiter and wrap with PAIR on the outside of the current expression.
		This command works fine in Erlang for the following code examples:	**Before:** `AList = [[1, |2, 3, 4]].` **After <M-f7> r {:** `AList = [{1, |2, 3, 4}]` **Before:** `AList = [[1, |2, 3, 4]].` **After C-u <M-f7> r {:** `AList = [{[1, |2, 3, 4]}]`
Swap current block and parent block wrapping characters • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> w**	(**sp-swap-enclosing-sexp** &optional ARG)	Swap the enclosing delimiters of this and the parent expression. • With N > 0 numeric argument, ascend that many levels before swapping.
		This command works fine in Erlang for the following code examples:	**Before:** `AList = ({[1, |2, 3, 4]}).` **After <M-f7> w:** `AList = ([{1, |2, 3, 4}]).` **Before:** `AList = ({[1, |2, 3, 4]}).` **After <M-f7> w:** `AList = [{(1, |2, 3, 4)}].`
Un-wrap block			
Extract all elements from current/next block • ∑𝔵 Smartparens **with smartparens-mode active** 	**<M-f7> U**	(**sp-unwrap-sexp** &optional ARG)	Un-wrap current or next block. • With ARG N, unwrap Nth expression as returned by 'sp-forward-sexp'. • If ARG is negative -N, unwrap Nth expression backwards as returned by 'sp-backward-sexp'
			Before: `AList = |({[1, 2, 3, 4]}).` **After <M-f7> U:** `AList = |[{1, 2, 3, 4}].` **Before:** `AList = ({[1, |2, 3, 4]}).` **After <M-f7> U:** `AList = ({1, |2, 3, 4}).` **Before:** `AList = [1, |2, [3, 4], 5, [6, 7], 8].` **After <M-f7> U:** `AList = [1, |2, 3, 4, 5, [6, 7], 8].` **Before:** `AList = [1, |2, [3, 4], 5, [6, 7], 8].` **After M-2 <M-f7> U:** `AList = [1, |2, [3, 4], 5, 6, 7, 8].`

Description	Keystroke	Function	Note
Extract all elements from previous block • ∑𝔵 Smartparens with smartparens-mode active	**<M-f7> W**	(**sp-backward-unwrap-sexp** &optional ARG)	Unwrap the previous block/expression. • With ARG N, unwrap Nth expression as returned by 'sp-backward-sexp'. • If ARG is negative -N, unwrap Nth expression forward as returned by 'sp-forward-sexp'. **Before:** AList = ({[1, |2, 3, 4]}). **After <M-f7> W:** AList = ({1, |2, 3, 4}). Again **After <M-f7> W:** AList = (1, |2, 3, 4). Again **After <M-f7> W:** AList = 1, |2, 3, 4. **Before:** AList = [0, 1, [2, |3, 4], 5]. **After <M-f7> W:** List = [0, 1, 2, |3, 4, 5]. **Before:** AList = [1, 2, [3, 4], 5, [6, 7], |8]. **After <M-f7> W:** AList = [1, 2, [3, 4], 5, 6, 7, |8]. **Before:** AList = [1, 2, [3, 4], 5, [6, 7], |8]. **After M-2 <M-f7> W:** AList = [1, 2, 3, 4, 5, [6, 7], |8].
Split & Join			
Split block • ∑𝔵 Smartparens with smartparens-mode active	**<M-f7> |**	(**sp-split-sexp** ARG)	Split the list or string the point is on into two. • If ARG is a raw prefix **C-u** split all the sexps in current expression in separate lists enclosed with delimiters of the current expression. smartparens by itself fails to process the first of these examples properly. PEL fixes the issues with post processing. **Before:** AList = [1, 2, [3, 4, |5, 6, 7], 8]. **After <M-f7> |:** AList = [1, 2, [3, 4], |[5, 6, 7], 8]. **Before:** Name = "Joe |Armstrong". **After <M-f7> |:** Name = "Joe "|"Armstrong". **Before:** AList = [1, 2, [3, 4, |5, 6, 7], 8]. **After C-u <M-f7> |:** AList = [1, 2, [3], [4], |[5], [6], [7], 8].
Join blocks • ∑𝔵 Smartparens with smartparens-mode active	**<M-f7> J**	(**sp-join-sexp** &optional ARG)	Join the blocks before and after point if they are of the same type. • If ARG is positive N, join N expressions after the point with the one before the point. • If ARG is negative -N, join N expressions before the point with the one after the point. • If ARG is a raw prefix **C-u** join all the terms up until the end of current expression. • The joining stops at the first expression of different type. **Before:** AList = [0, 1, [2, 3, 4]|, [5, 6], 7]. **After <M-f7> J:** AList = [0, 1, [2, 3, 4|, 5, 6], 7]. **Before:** AList = [[0, 1]|, [2, 3, 4]|, [5, 6], 7]. **After M-2 <M-f7> J:** AList = [[0, 1|, 2, 3, 4, 5, 6], 7].
Search Support	In Erlang 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 Erlang mode. To change this use the **<f11> t <f2>** to access the customize buffer.		
Toggle superword-mode • ∑ Text Modes • ∑ Search/Replace	**<f12> M-p** • **<f11> t m p** • **<f11> SPC e M-p**	(**superword-mode** &optional ARG)	Toggle superword-mode: a minor mode that treats **snake_case** as one word. • In Erlang, '_' are then treated as part of words. • With prefix argument ARG, enable superword mode if ARG is positive, disable it otherwise.
Highlighting blocks	The following commands can be used to activate or toggle useful modes to highlight blocks of (), {}, and []. • show-paren-mode, which highlights the parens that matches the one before or after point. • rainbow delimiters mode, where matching nested parens are highlighted with the same colour.		
Toggle show-paren mode on/off See also: ∑ Highlight	• **<f12> M-9** • **<M-f12> M-9** • **<f11> h (** • **<f11> SPC e M-9**	(**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.
Toggle colouring of nested blocks See also: ∑ Highlight	• **<f12> M-r** • **<M-f12> M-r** • **<f11> h R**	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** **Requires:** rainbow-delimiters.el activated by **pel-use-rainbow-delimiters**.
Edit Erlang Code	The following commands help edit Erlang code.		
Create additional clause	**C-c C-j**	(**erlang-generate-new-clause**)	Create additional Erlang clause header. • Parses the source file for the name of the current Erlang function. Create the header containing the name, a pair of parentheses, and an arrow. The space between the function name and the first parenthesis is preserved. The point is placed between the parentheses.
Clone clause arguments	**C-c C-y**	(**erlang-clone-arguments**)	Insert, at the point, the argument list of the previous clause. • Copy the function arguments of the preceding Erlang clause. This command is useful when defining a new clause with almost the same argument as the preceding. • The mark is set at the beginning of the inserted text, the point at the end.
Insert Erlang Code with Specialized Tempo Skeletons *Erlang Style Control* ☛ See also: • ∑ Inserting Text for more info and information about tempo skeleton and the completely different yasnippet template-based text insertion). **+** : additional templates **C** : templates with customization control	The erlang.el external package defines a set of text skeletons, available on the Erlang/Skeletons menu (via **<f10>**) • PEL provides the following additional functionality: • Quick access keys to insert the templates, all mapped under the **pel:erlang-skel** key prefix: **<f12> <f12>**. • Several additional templates. These are marked with a **+**. These are also added to the menu. • Several aspects of the PEL Erlang Source Code Style is controlled by the user options inside the **pel-erlang-code-style** group. The controlled templates affected are marked with a **C**. The relevant user options are part of the **pel-erlang-code-style** group accessible with <f12> <f2> from an erlang mode buffer and include the following options: • **pel-erlang-skel-insert-file-timestamp** : set whether an automatically updated timestamp is inserted in the file header block. • **pel-erlang-skel-prompt-for-purpose** : set whether file and function skeletons blocks prompt for purpose and insert it. • **pel-erlang-skel-prompt-for-function-name** : set whether function skeletons prompt for function name and then inserts that name. • **pel-erlang-skel-prompt-for-function-arguments** : set whether function skeletons prompt for function arguments and then insert them. • **pel-erlang-use-separators** : set whether blocks use horizontal separator lines (these are the first of potentially 2 separators). • **pel-erlang-use-secondary-separators** : set whether blocks use a second block horizontal separator line. • **pel-erlang-skel-with-edoc** : set whether generated code comments use EDoc markup. • **pel-erlang-skel-with-license** : set whether file header blocks use open source software license text controlled by lice. ☝ Emacs user options by default take effect globally. But by using file and directory variables (see ∑ File/Directory Variables) they can also be used to take effect on a single file or all files inside a directory tree. So by default, the user options that control the PEL tempo template take effect globally. If you want to change the behaviour for only one file, write the user option control block at the end of that file. If you want to control the behaviour of the PEL tempo templates for all files inside a directory tree create a .dir-locals file and store the values of the relevant options variables inside that file. This allows you to control the user options affecting the format of the tempo templates precisely and does not affect what you actually type. • Once a skeleton was just entered (or later by activating the pel-tempo-mode) you can move to the next or previous point of interest (so called *tempo-marks)* with the standard tempo-mode keys **C-c M-f** and **C-c M-b** or some other keys like **C-c .** and **C-c ,**. • Instead of using the **<f12> <f12>** bindings, you can also type the template name and then hit **C-c C-M-i** or **<f12> <f12> <f12>**. This supports listing all completions into a separate temporary buffer. This is mainly useful for templates which short names such as "if", "case", etc… ☝ Some of the template names in the title column are also links to the relevant Erlang language construct reference page.		

Description	Keystroke	Function	Note
∑ **Customize** PEL Erlang Skeletons layout	`<f12> <f12> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL Erlang skeleton layout. • If OTHER-WINDOW is non-nil (use `C-u`), display in another window.
if	`<f12> <f12> i`	(**pel-erl-if**)	Insert an if statement.
case	`<f12> <f12> c`	(**pel-erl-case**)	Insert a case expression.
export +	`<f12> <f12> x`	(**pel-erl-export**	Insert an export module attribute expression.
import +	`<f12> <f12> I`	(**pel-erl-import**)	Insert an import module attribute expression.
try +	`<f12> <f12> t`	(**pel-erl-try**)	Insert a try expression.
try-of +	`<f12> <f12> T`	(**pel-erl-try-of**)	Insert a try expression with of clauses.
receive	`<f12> <f12> r`	(**pel-erl-receive**)	Insert a receive expression.
after	`<f12> <f12> a`	(**pel-erl-after**)	Insert a receive expression with an after (timeout) clause.
loop	`<f12> <f12> l`	(**pel-erl-loop**)	Insert a simple receive loop.
module	`<f12> <f12> m`	(**pel-erl-module**)	Insert the module attribute.
function C	`<f12> <f12> f`	(**pel-erl-function**)	Insert a function definition. This may prompt for function name, argument and purpose according to the user options described above. All prompts maintain independent histories.
author	`<f12> <f12> ` `	(**pel-erl-author**)	Insert the author attribute. Uses the **user-mail-address** user option to insert your mail address.
spec	`<f12> <f12> s`	(**pel-erl-spec**)	Insert a **-spec** for the function following point.
small-header C	`<f12> <f12> M-h`	(**pel-erl-small-header**)	Insert a small file header without any comment.
normal-header C	`<f12> <f12> M-H`	(**pel-erl-normal-header**)	Insert a normal file header: includes author name, copyright notice, doc section, file created date
large-header C	`<f12> <f12> h`	(**pel-erl-large-header**)	Insert a large header block that includes all normal header fields plus separators. • User-options control the format. Distinguish Erlang .erl module files from the .hrl header files.
small-server C	`<f12> <f12> M-s`	(**pel-erl-small-server**)	Insert a large file header and template logic for a small server.
application C	`<f12> <f12> M-a`	(**pel-erl-application**)	Insert a large file header and template logic for an **application behaviour**.
supervisor C	`<f12> <f12> M-u`	(**pel-erl-supervisor**)	Insert a large file header and template logic for a **supervisor behaviour**.
supervisor-bridge C	`<f12> <f12> M-b`	(**pel-erl-supervisor-bridge**)	Insert a large file header and template logic for a **supervisor bridge behaviour**.
generic-server C	`<f12> <f12> M-g`	(**pel-erl-generic-server**)	Insert a large file header and template logic for a **gen-server behaviour**.
gen-event C	`<f12> <f12> M-e`	(**pel-erl-gen-event**)	Insert a large file header and template logic for a **gen-event behaviour**.
gen-fsm C	`<f12> <f12> M-f`	(**pel-erl-gen-fsm**)	Insert a large file header and template logic for a **gen-fsm behaviour**.
gen-statem-StateName C	`<f12> <f12> M-S`	(**pel-erl-gen-statem-StateName**)	Insert a large file header and template logic for a **gen-statem behaviour**.
gen-statem-handle-event C	`<f12> <f12> M-E`	(**pel-erl-gen-statem-handle-event**)	Insert a large file header and template logic for a gen-statem.
wx-object C	`<f12> <f12> M-w`	(**pel-erl-wx-object**)	Insert a large file header and template logic for a wx-object generic server.
gen-lib C	`<f12> <f12> M-l`	(**pel-erl-gen-lib**)	Insert a large file header and template logic for a library module.
gen-corba-cb C	`<f12> <f12> M-c`	(**pel-erl-gen-corba-cb**)	Insert a large file header and template logic for a **CORBA** callback module.
ct-test-suite-s	`<f12> <f12> M-1`	(**pel-erl-ct-test-suite-s**)	Insert a large file header and template logic for a test suite
ct-test-suite-l	`<f12> <f12> M-2`	(**pel-erl-ct-test-suite-l**)	Insert a large file header and template logic for a test suite
ts-test-suite	`<f12> <f12> M-3`	(**pel-erl-ts-test-suite**)	Insert a large file header and template logic for a test suite
Tempo Template Tag Insertion	• `C-c C-M-i` • `<f12> <f12> <f12>`	(**tempo-complete-tag** &optional SILENT)	Look for a tag and expand it. ☝Instead of using the `<f12> <f12>` key bindings above, type the template name and then hit `C-c C-M-i`. (or `<f12> <f12> <f12>`).
	A completion buffer opens up if the template name is incomplete (or empty in which case the buffer lists **all** available template names). Select the template name and hit RET. Emacs expands the template. • All the tags in the tag lists in 'tempo-local-tags' (including 'tempo-tags') are searched for a match for the text before the point. The string matching is determined by the variable 'tempo-match-finder'. If 'tempo-match-finder' returns nil, then the results are the same as no match at all. • If a single match is found, the corresponding template is expanded in place of the matching string. • If a partial completion or no match at all is found, and SILENT is non-nil, the function will give a signal. • If a partial completion is found and 'tempo-show-completion-buffer' is non-nil, a buffer containing possible completions is displayed.		
Toggle pel-tempo-mode See also: • ∑ **Inserting Text**	`<f12> <f12> SPC` • `<f11> SPC e <f12> SPC` • `<f6> SPC`	(**pel-tempo-mode** &optional ARG)	Toggle PEL tempo mode on/off. PEL tempo mode activates `C-c .` and `C-c ,` as well as `C-c C-.` and `C-c C-,` key bindings to navigate across tempo mark hot-spots. When pel-tempo-mode is active the pel-tempo-mode lighter (‡) is shown on the status bar. The second set are only available when Emacs runs in graphics mode. ☝When a skeleton is inserted via the execution of one of the pel-erl-… commands above, the pel-tempo-mode is automatically activated.
Jump to next tempo mark	• `C-c M-f` • `C-c .` • `C-c C-.`	(**tempo-forward-mark**)	Jump to the next mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key key bindings are only available when pel-tempo-mode is active.
Jump to previous tempo mark	• `C-c M-b` • `C-c ,` • `C-c C-,`	(**tempo-backward-mark**)	Jump to the previous mark in 'tempo-back-mark-list': the location where code must be updated inside the inserted skeleton. • These key binding are only available when pel-tempo-mode is active.

Description	Keystroke	Function	Note
Specialized Kill See also: • ∑ Cut & Paste • ∑𝓍 Smartparens	Specialized delete and kill commands are provided by the 📦 The **smartparens** external package 🆙 activated by **pel-use-smartparens** user-option. • Activate smartparens mode manually with **<f11> (** or automatically by adding smartparens-mode to **pel-erlang-activates-minor-mode**. • This table uses the ⌦ and ⌫ symbols to represent these 2 keys: ⌦ := "forward delete" := **<deletechar>** := **Fn** ⌫ ⌫ := "backward delete" := **<backspace>** Often labelled "delete" on keyboards. • With PEL the commands that are marked with 👁 display the killed string when **pel-show-copy-cut-text** is **t**. Toggle this display with **<f11> M-=**		
• **Delete char**	When **smartparens** is used, the delete keys protect deletion of balanced pairs but allow deletion of marked areas regardless of the block pairs.		
Standard delete forward character	• **<deletechar>** • ⌦	(**delete-forward-char** N &optional KILLFLAG)	Delete the following N characters (previous if N is negative). • If Transient Mark mode is enabled, the mark is active, and N is 1, delete the text in the region and deactivate the mark instead. • To disable this, set variable 'delete-active-region' to nil. Interactively, N is the prefix arg, and KILLFLAG is set if N was explicitly specified. • When killing, the killed text is filtered by 'filter-buffer-substring' before it is saved in the kill ring, so the actual saved text might differ from the killed text.
Delete forward, jump over block pair until block is empty then delete block 🎛 • ∑𝓍 Smartparens **with smartparens-mode active**	• **<deletechar>** • ⌦	(**pel-sp-delete-char** &optional ARG)	Same as above with the **additional behaviour** listed below. 💻 Execute 'sp-delete-char' if no area marked, otherwise delete marked area. • If an area is marked: • deletes the area, regardless of the presence of blocks, even if the resulting text would lead to unbalanced pairs. It ignores the prefix argument and the state of delete-selection-mode. • If nothing is marked: • If on an opening delimiter, move forward into balanced expression. • If on a closing delimiter, refuse to delete unless the balanced expression is empty, in which case delete the entire expression. • If the delimiter does not form a balanced expression, it will be deleted normally. • With a numeric prefix argument N = 0, simply delete a character forward, without regard for delimiter balancing. • If ARG is raw prefix argument **C-u**, delete characters forward until a closing delimiter whose deletion would break the proper pairing is hit.
Standard delete backward character	• **DEL** • ⌫	(**backward-delete-char-untabify** ARG &optional KILLP)	Delete characters backward, changing tabs into spaces. • Delete ARG chars, and kill (save in kill ring) if KILLP is non-nil. • Interactively, ARG is the prefix arg (default 1) and KILLP is t if a prefix arg was specified. • The exact behavior depends on 'backward-delete-char-untabify-method'.
Delete character - backward, jump over block pair until block is empty then delete block 🎛 • ∑𝓍 Smartparens **with smartparens-mode active**	• **DEL** • ⌫	(**pel-sp-backward-delete-char** &optional ARG)	Same as above with the **additional behaviour** : • If an area is marked deletes the area, regardless of the presence of blocks, even if the resulting text would lead to unbalanced pairs. It also ignores the prefix argument. • When nothing is marked: • Deletes character before cursor (deletes backward), replaces hard tab with spaces as required. • Does not delete only one side of a balanced pair block: instead move into the block and delete its content until it is empty. When the block is empty the command deletes both block characters. • If on a closing delimiter, move backward into balanced expression. • If on an opening delimiter, refuse to delete unless the balanced expression is empty, in which case delete the entire expression. • If the delimiter does not form a balanced expression, it will be deleted normally. • With a numeric prefix argument N = 0, simply delete a character backward, without regard for delimiter balancing. If ARG is raw prefix argument **C-u**, delete characters backward until an opening delimiter whose deletion would break the proper pairing is hit. 💻 Execute 'sp-delete-char' if no area marked, otherwise delete marked area.
• **Delete char** • Does not delete marked areas with balanced pairs.	The following commands are long-winded key sequences that delete forward and backward without breaking blocks. • The forward and backward delete keys do the same when smartparens-mode is active. • ☹️ Note that these will **not** accept to delete or kill a region that contains balanced pairs even if the region contains the two sides! • This is why PEL maps the standard delete forward and backwards keys commands that use smart-parens delete as long as the area is not marked.		
Delete char forward	**<M-f7> DEL n**	(**sp-delete-char** &optional ARG)	Delete a character forward or move forward over a delimiter. • If on an opening delimiter, move forward into balanced expression. • If on a closing delimiter, refuse to delete unless the balanced expression is empty, in which case delete the entire expression. • If the delimiter does not form a balanced expression, it will be deleted normally. • With a numeric prefix argument N > 0, delete N characters forward. • With a numeric prefix argument N < 0, delete N characters backward. • With a numeric prefix argument N = 0, simply delete a character forward, without regard for delimiter balancing. • If ARG is raw prefix argument C-u, delete characters forward until a closing delimiter whose deletion would break the proper pairing is hit. `(quu|x "zot") -> (quu| "zot")` `(quux |"zot") -> (quux "|ot")` `(foo (|) bar) -> (foo | bar)` `|(foo bar)  -> (|foo bar)`
Delete char backward	**<M-f7> DEL p**	(**sp-backward-delete-char** &optional ARG)	Delete a character backward or move backward over a delimiter. • It has the same description as the above command but goes backward instead of forward. `("zot" q|uux) -> ("zot" |uux)` `("zot"| quux) -> ("zot|" quux) -> ("zo|" quux)` `(foo (|) bar) -> (foo | bar)` `(foo bar)|  -> (foo bar|)`
• **Delete/Kill region**	The following commands delete marked regions as long as the deletion would not create unbalanced blocks. • These may be useful inside keyboard macros when deleting text in area where several balanced and nested blocks are present. • ☹️ Note that these will **not** accept to delete or kill a region that contains balanced pairs even if the region contains the two sides!		
Delete region	**<M-f7> DEL -**	(**sp-delete-region** BEG END)	Delete the text between point and mark, like 'delete-region'. • BEG and END are the bounds of region to be deleted. • If that text is unbalanced, signal an error instead. • With a prefix argument, skip the balance check.
Kill region	**<M-f7> - -**	(**sp-kill-region** BEG END)	Kill the text between point and mark, like 'kill-region'. • BEG and END are the bounds of region to be killed. • If that text is unbalanced, signal an error instead. • With a prefix argument, skip the balance check.
• **kill block elements**	The following commands kill the element(s) of a block.		
Kill content of next block 👁 • ∑𝓍 Smartparens	• **<M-f7> ⌦** • **<M-f7> - n**	(**sp-change-inner**)	Change the content of current or next block. Point can be anywhere in block or element before block. **Before:** `{'EXIT',Reason} ->` `  {|error,{asn1,Reason}};` **After:** `{'EXIT',Reason} ->` `  {error,{|}};`
Delete content of current block 👁 • ∑𝓍 Smartparens	**<M-f7> - .**	(**sp-change-enclosing**)	Delete content of the enclosing block. Point can be anywhere inside the current block. **Before:** `{'EXIT',Reason} ->` `  {error,{|asn1,Reason}};` **After:** `{'EXIT',Reason} ->` `  {error,{|}};`
Kill block elements forward 👁 • ∑𝓍 Smartparens	**<M-f7> -]**	(**sp-kill-sexp** &optional ARG DONT-KILL)	Kill block elements after point. **Before:** `case Tlv9 of` `  [] -> true;_ -> exit({error,|{asn1, {unexpected, Tlv9}}})` **After:** `case Tlv9 of` `  [] -> true;_ -> exit({error,|})`

Description	Keystroke	Function	Note
Kill block elements backward • ∑𝓍 Smartparens	**<M-f7> - [**	(**sp-backward-kill-sexp** &optional ARG DONT-KILL)	Kill block elements before point. **Before:** `case Tlv9 of` `  [] -> true;_ -> exit({error,|{asn1, {unexpected, Tlv9}}})` **After:** `case Tlv9 of` `  [[] -> true;_ -> exit({|{asn1, {unexpected, Tlv9}}})`
Kill element after current • ∑𝓍 Smartparens	**<M-f7> - }**	(**sp-kill-hybrid-sexp** ARG)	Kill a line as if with 'kill-line', but respecting delimiters. • With ARG being raw prefix **C-u C-u**, kill the hybrid sexp the point is in (see 'sp-get-hybrid-sexp'). • With ARG numeric prefix 0 (zero) just call 'kill-line'. • You can customize the behaviour of this command by toggling 'sp-hybrid-kill-excessive-whitespace'.
Kill whole line	**<M-f7> - l**	(**sp-kill-whole-line**)	⚠️ Currently this deletes the whole line. Requires Erlang specific implementation. 🚧
• Kill/splice			
Un-wrap current block, splicing its elements in enclosing block • ∑𝓍 Smartparens	**<M-f7> l l**	(**sp-splice-sexp** &optional ARG)	Un-wrap current block, splicing its content in enclosing block (if any). **Before:** **After:** `{|EncBytes,EncLen} = 'enc'(Cdx, []),  |EncBytes,EncLen = 'enc'(Cdx, []),` **Before:** `-asn1_info(` ` [{vsn,'2.0.1'},` `  {module,'ELDAPv3'},` `  {options,[{i,"src"},{|outdir,"src"},noobj,{i,"."},{i,"asn1"}]}]).` **After:** `-asn1_info(` ` [{vsn,'2.0.1'},` `  {module,'ELDAPv3'},` ` {options,[{i,"src"}, |outdir,"src",noobj,{i,"."},{i,"asn1"}]}]).
Kill block element(s) before point and splice remaining into outer block • ∑𝓍 Smartparens	**<M-f7> l [**	(**sp-splice-sexp-killing-backward** &optional ARG)	Kill elements before point in block and splice remaining elements into outer block. **Before:** `case Tlv9 of` `  [] -> true;  -> exit({error,{asn1, {unexpected, |Tlv9}}})` **After:** `case Tlv9 of` `  [] -> true;  -> exit({error,{asn1, |Tlv9}})`
Kill block element(s) forward and splice remaining into outer block • ∑𝓍 Smartparens	**<M-f7> l]**	(**sp-splice-sexp-killing-forward** &optional ARG)	Kill elements after point in block and splice remaining elements into outer block. **Before:** `case Tlv9 of` `  [] -> true;  -> exit({error,{asn1, {unexpected, |Tlv9}}})` **After:** `case Tlv9 of` `  [] -> true;  -> exit({error,{asn1, unexpected|}})`
Kill around element • ∑𝓍 Smartparens	**<M-f7> l o**	(**sp-splice-sexp-killing-around** &optional ARG)	Kill content around current element/block. **Before:** `-asn1_info(` ` [{vsn,'2.0.1'},` `  {module,'ELDAPv3'},` `  {options,[{i,"src"},|{outdir,"src"},noobj,{i,"."},{i,"asn1"}]}]).` **After:** `-asn1_info(` ` [{vsn,'2.0.1'},` `  {module,'ELDAPv3'},` ` {options,|{outdir,"src"},}]).
• Delete/Kill word	These commands complements the standard word kill commands normally available with shorter key bindings. See ∑ Cut & Paste		
Delete word backward	**<M-f7> DEL v**	(**sp-backward-delete-word** &optional ARG)	(sp-backward-delete-word &optional ARG) • Delete a word backward, skipping over intervening delimiters. • Deleted word does not go to the clipboard or kill ring. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction.
Delete word forward	**<M-f7> DEL w**	(**sp-delete-word** &optional ARG)	Delete a word forward, skipping over intervening delimiters. • Deleted word does not go to the clipboard or kill ring. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction.
Kill word backward	**<M-f7> - v**	(**sp-backward-kill-word** &optional ARG)	Kill a word backward, skipping over intervening delimiters. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction. 🚧
Kill word forward	**<M-f7> - w**	(**sp-kill-word** &optional ARG)	Kill a word forward, skipping over intervening delimiters. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction.
• Delete/Kill symbol	See '**sp-backward-symbol**' and '**sp-forward-symbol**' for what constitutes a symbol for the backward and forward commands respectively. • These commands complements the standard word delete commands normally available with shorter key bindings. See ∑ Cut & Paste		
Delete symbol backward	**<M-f7> DEL a**	(**sp-backward-delete-symbol** &optional ARG WORD)	Delete a symbol backward, skipping over any intervening delimiters. • Deleted symbol does not go to the clipboard or kill ring. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in forward direction.
Delete symbol forward	**<M-f7> DEL s**	(**sp-delete-symbol** &optional ARG WORD)	Delete a symbol forward, skipping over any intervening delimiters. • Deleted symbol does not go to the clipboard or kill ring. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction.
Kill symbol backward	**<M-f7> - a**	(**sp-backward-kill-symbol** &optional ARG WORD)	Kill a symbol backward, skipping over any intervening delimiters. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in forward direction.
Kill symbol forward	**<M-f7> - s**	(**sp-kill-symbol** &optional ARG WORD)	Kill a symbol forward, skipping over any intervening delimiters. • With ARG being positive number N, repeat that many times. • With ARG being Negative number -N, repeat that many times in backward direction.

Description	Keystroke	Function	Note
Erlang syntax checking Using either: • flycheck or • flymake See also: • ∑ SyntaxCheck			🎛 Syntax checking for the Erlang programming language can be done with Emacs built-in **flymake** as well as with the 📦 **flycheck** external package. • To activate either set the **pel-use-erlang-syntax-check** user option is set to either 'use-flycheck or 'use-flymake. • By default, the syntax checker is not automatically launched. If you want to start your selected syntax checker as soon as any Erlang file is opened, add 'erlang-mode to the **pel-modes-activating-syntax-check** user-option. • **flymake** is built-in Emacs. The Emacs erlang package provides erlang-flymake to use with Erlang. • PEL automatically installs and activates **flycheck** when **pel-use-erlang-syntax-check** user option is set to 'use-flycheck. 🎛 Flymake has several customizable variables, which some listed here: The following customization variables determine the exact circumstances whereupon Flymake decides to initiate a check of the buffer: • **flymake-start-on-flymake-mode** : **t** to start checking when flymake-mode is started. **nil** to prevent check. • **flymake-no-changes-timeout** : time to wait after last change to start checking. Default = 0.5 seconds. • **flymake-start-syntax-check-on-newline** : **t** to check after insertion or removal of newline char from buffer. **nil** to prevent check. The following variable control navigation to next or previous error: • **flymake-wrap-around** : If non-nil, moving to errors wraps around buffer boundaries. • **flymake-diagnostic-types-alist** : Alist ((KEY . PROPS)*) of properties of Flymake diagnostic types. See Emacs documentation for more info. The **M-n** and **M-p** keys are mapped to flymake commands only when flymake-mode is turned on.
Activate/deactivate selected syntax checker	**<f12> !** **<f11> SPC e !**	(**pel-erlang-toggle-syntax-checker**)	Toggle the selected Erlang syntax checker mode on/off. • The syntax checker activated or deactivated is either flycheck or flymake, as selected by the user-option variable `**pel-use-erlang-syntax-check**'. 📦🎛 See the required settings above to activate this command and select the syntax checker.
Go to next flymake diagnostic	**M-n**	(**flymake-goto-next-error** &optional N FILTER INTERACTIVE)	Move point to the next Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.
Go to previous flymake diagnostic	**M-p**	(**flymake-goto-prev-error** &optional N FILTER INTERACTIVE)	Move point to the previous Flymake diagnostic. • With a prefix arg, skip any diagnostics with a severity less than ':warning'. • Display the error message in the echo line.
Compiling Erlang Code	The following commands are used to compile Erlang source code files to .beam files located in the same directory as the source code. Detected errors are listed in the *erlang* shell opened to compile the files. The buffer shows the location of error and the error description. The following commands are used to navigate to the next or previous detected error.		
Compile code	• **C-c C-k** • **<f12> M-c** • **<M-f12> M-c**	(**erlang-compile**)	Compile Erlang module in current buffer. • If buffer visiting file was modified and not saved, prompts the user to save it first. • Opens and *erlang* shell, in which the Erlang compile is done with a eshell c() command. • The buffer lists the errors. Hitting **RET** on the error file/line move point to that line in the Erlang file buffer. The **RET** key is bound to (**compile-goto-error** &optional EVENT) • It's also possible to use the next-error and previous error.
Display compilation output	**C-c C-l**	(**erlang-compile-display**)	Display compilation output. • Essentially opens the shell buffer where the last compilation occurred. If that shell was closed nothing can be displayed.
Move to next compile error	• **C-x `** • **M-g n** • **M-g M-n**	(**next-error** &optional ARG RESET)	A prefix ARG specifies how many error messages to move; • negative means move back to previous error messages. • Just **C-u** as a prefix means reparse the error message buffer and start at the first error. ⚠ This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must compile the file first.
Move to previous compile error	• **M-g p** • **M-g M-p**	(**previous-error** &optional N)	Prefix arg N says how many error messages to move backwards (or forwards, if negative). ⚠ This only shows the result of compilations; it does not report Flycheck reported errors. To use it you must compile the file first.
Move to next compilation or Flycheck detected error	**C-c C-n**	(**edts-code-next-issue** &optional WRAPPED)	Moves point to the next error in current buffer and prints the error. ☝ When Flymake is active, this command can be used as soon as an error is reported, even if the file was not compiled.
Move to previous compilation or Flycheck detected error	**C-c C-p**	(**edts-code-previous-issue** &optional WRAPPED)	Moves point to the next error in current buffer and prints the error. ☝ When Flymake is active, this command can be used as soon as an error is reported, even if the file was not compiled.
Development Tool	The following commands are used when adding Emacs Lisp support for Erlang.		
Show syntactic information	**C-c C-s**	(**erlang-show-syntactic-information**)	Show syntactic information for current line. • Display semantic Lisp data structure in the echo line. Not useful for writing Erlang.
Erlang Shell	Commands to explicitly launch or re-open an Erlang shell that runs under an Emacs inferior-erlang process controlled by the comint mode from the comint.el library running in erlang-shell-mode.		
Open Erlang Shell	**C-c C-z**	(**erlang-shell-display**)	Display the existing Erlang shell, or start a new. Available from Erlang mode buffers only.
Start new Erlang Shell	**<f11> z r e** **<f12> z**	(**erlang-shell**)	Start a new Erlang shell. Can be used from any buffer. • The variable 'erlang-shell-function' decides which method to use, default is to start a new Erlang host. It is possible that, in the future, a new shell on an already running host will be started. • **C-c C-z** starts the Erlang Shell from the Erlang Mode. • **<f11> z r** is available globally and will work as long as the erl executable is accessible. 🎛 Under PEL this command is available only when the **pel-use-erlang** user option is set to **t**.
Work around to issues in the Erlang Shell	When running the **Erlang Shell** inside Emacs, you may run into some issues. They are listed here along with work-arounds. • *Redundant command echo*: On some systems the Erlang shell annoyingly echoes each typed command. If this is the case for your system, PEL provides a fix: 🎛🎛 Set the **pel-erlang-shell-prevent-echo** user option to **t**. After doing that execute pel-init or restart Emacs. • *Typing Ctrl-G does not open the* **Erlang JCL Command Menu**: work-around: type the following instead: **C-q C-g RET** ⚠ Unfortunately the above workaround does not work when the Erlang shell is launched inside an Emacs vterm shell (see ∑ **Shells**).		
Erlang Shell: Command History	The following commands can be used to retrieve previously issued Erlang shell commands at the shell prompt. ☝ **Erlang shell command history file**: • The Erlang shell history controlled by Emacs is saved inside a file the is restored when opening a new shell: commands from previously opened Erlang shells are also available. • Within an Emacs inferior-erlang the • You can also use the **Erlang shell commands** to access the local shell history.		
Next shell command	**M-n**	(**comint-next-input** ARG)	Cycle forwards through Erlang shell input history.
Previous shell command	**M-p**	(**comint-previous-input** ARG)	Cycle backwards through Erlang shell input history, saving input.

Description	Keystroke	Function	Note
Using Man inside Emacs and support Erlang Man pages See also: ∑ **Help/Info** **About Erlang** See also: ∑ **Menus**	Emacs provide 2 main commands to display man pages inside buffers. • Both of these are much more powerful than the usual man reader available on the shell allowing navigation across man pages and opening hyperlinks. They are: • The man command uses the system man utility • WoMan: Browse Unix Manual Pages "W.O. (without) Man" a complete implementation. It has some formatting limitations compared to man but it's very useful in systems where man is not available like Windows. **To see Erlang man pages using the man command**: On most systems the Man pages for Erlang are not available to the man utility and therefore not available for man inside Emacs. There are several ways this can be remedied: • One is to set the MANPATH environment variable to include the directory where these files are located. Then man can be used outside and inside Emacs to access Erlang's man pages. For example the following lines can be stored inside a shell script to do this: `MANPATH=/usr/local/Cellar/erlang/22.3.4/lib/erlang/man:`manpath` export MANPATH` • Another way is to customize the Emacs **Man-switches** user option variable to something that includes the same directory. This will add the capability of Emacs man to fin the Erlang's man pages without modifying the capabilities of the parent shell. For example, if we want to use the same directory as the above example we need to set the Man-switches which is normally set to nil to the following value: "-M`manpath`:/usr/local/Cellar/erlang/22.3.4/lib/erlang/man" The second alternative can be used to add other directories for the man pages of other programming languages while leaving the ability to have several shells that have their own value of MANPATH. That might be very useful for someone that uses different versions of Erlang in a system and needs access to the man pages of different versions of Erlang. It becomes possible to run different shells inside Emacs with each having its own value of MANPATH and therefore providing the man pages from different locations. It is also possible to place all of these directories inside the Man-switches or MANPATH and buses man's ability to view several pages for the same topic. **To only see Erlang topics in Man completion:** When learning Erlang it might help to see only Erlang topics when using the man command completion. To do that , set MANPATH to the Erlang man directory only. You must also ensure that a whatis file is located in the Erlang man page root directory, otherwise Emacs man completion will not work. See my description on how to create whatis file for local man directory. **Using EDTS to access the man pages of the version of Erlang used by various projects**: EDTS (see below) supports the ability to download and access man pages of several Erlang versions, tied to your Erlang projects. EDTS provides it's own help command to access sections inside the mane pages, allowing EDTS driven man page access to co-exist with manual man command execution and the techniques described above. **PEL supports multiple versions of Erlang and access to their man pages** Inside the **pel-erlang-environment** group, the **pel-erlang-man-parent-rootdir** user-option can be set to read the man parent directory name from an environment variable. To support the ability to open the man files related to a specific version of Erlang available to the parent OS shell, set the environment variable when you select the version of Erlang available to the OS shell and set the name of the environment variable in the **pel-erlang-man-parent-rootdir** user-option. See the following **Installing Erlang** pages of the **About Erlang** document that describes an setting such an editing environment: • **Install Erlang OTP Documentation and Man Files** • **Creating whatis files for Erlang man pages** • **Using the Erlang Man files within Emacs** • **Using Specialized OS Shells for Erlang** • **Using PEL with Specialized Shells for Erlang to Edit Erlang** Use the following commands to open an Erlang man page inside Emacs. • You can also use the toolbar menu (with PEL open it with **<f10>**) in the Erlang section.		
Open a man page inside an Emacs buffer See also: • ∑ **Help/Info** • ∑ **Customize**	• **<f11> ? m** • ⌘**-M**	(**man** MAN-ARGS)	Using man pages inside emacs is even better than using it from the shell because: • the links are active and can be followed. When the man page describes a directory or file, emacs will open the file or the directory (in direct mode) when pressing **RET** over the link. • You can navigate easily between sections (n/p will move to the next/previous section) • You can use any of the searches. • You can use any of the options to the man command at the prompt, like the -a option to access all man pages of the same name. Then use **M-n** and **M-p** to move from one to the other page, inside the same buffer. • See all keys available in mode, with **<f1> m** or **<f11> ? k m**. ☝ The man command prompts, using the word at point as the default. 🎛 PEL key sequence to customize man: **<f11> <f2> E m**
Open a man page without external man process: woman • ∑ **Help/Info** • ∑ **Customize**	**<f11> ? w**	(**woman** &optional TOPIC RE-CACHE)	Open a man page file in Emacs using the woman mode, completely implemented in Emacs Lisp (and therefore without using the external 'man' process). That can be very useful under environments where man is not available (such as basic Windows). 🎛 PEL key sequence to customize man: **<f11> <f2> E w** • text width, use word at point, etc…
Show Manual page for Erlang function at point	• **C-c C-d** • **<M-f12> M-d**	(**erlang-man-function-no-prompt**)	Find manual page for the function under the cursor. The man entry for 'function' is displayed. This function provides the same functionality as erlang-man-function except for that it does not ask the user to confirm the function name before opening the man page for the function.

Description	Keystroke	Function	Note
EDTS	**EDTS - Erlang Development Tool Suite** 📦 The commands in the following rows require the EDTS external package. PEL activates it when the **pel-use-edts** user option is set to **t**. • If you want EDTS to start automatically when you open an Erlang file, set **pel-use-edts** to **start-automatically** instead of **t**. ⚠️ If EDTS reports that it cannot start the main server, check the ~/.emacs.d/elpa/edts-XXX directory. It should contain a _build sub-directory created by the Makefile. If that does not exists, open a shell that has access to Erlang, cd to ~/.emacs.d/elpa/edts-XXX and type **make** to build what is missing. • See **EDTS issue 145**.		
Erlang Project settings See also: **Sessions**	🎛 EDTS is customizable through it **edts** customization group. • With PEL, you can access it by typing **<f12> <f3>** from an Erlang buffer, or **<f11> SPC e <f3>** from any other buffer) and type character that identified edts. • EDTS also uses an external **.edts** configuration file to store Erlang project specific settings. See **EDTS: Configure your projects.** This allows setting the following: project name, node-name, erlang-cookie, lib-dirs, start-command, top-path, dialyzer-plt, app-include-dirs, project-include-dirs, xref-error-whitelist, xref-file-whitelist ⚠️ Desktop restoration often fails when edts-mode was active on session stored: unfortunately edts does not provide a desktop restore handler. • ☝ PEL does, however provide a desktop restore handler for EDTS which detects edts-mode failures and protect the desktop restoration. ☛ If EDTS has not been activated yet, the only EDTS specific key available is **<f12> M-E** to activate it. Once it's activated the other keys are available.		
Toggle EDTS mode	**<f12> M-E** **<f11> SPC e M-E**	(**edts-mode** &optional ARG)	Turn EDTS mode on or off. • EDTS is an easy to set up Development-environment for Erlang. • EDTS also incorporates a couple of other minor-modes, currently auto-highlight-mode and auto-complete-mode. They are configured to work together with EDTS but see their respective documentation for information on how to configure their behaviour further.
EDTS/Navigation	**EDTS** (see below) provides 2 commands to move point across Erlang functions: ferl-goto-previous-function and ferl-goto-next-function. They are listed above in the navigation section. The EDTS navigation functions do not support repetition prefix argument nor they support shift marking. There are other commands and key bindings to move across Erlang functions, and PEL support functions that perform the same and support repetition and shift marking. See the commands listed in the navigation section above.		
EDTS/AHS Editing	EDTS supports the automatic highlight symbol mode (AHS) and provides commands to modify the name of the highlighted name in the current function or in all of the buffer. The automatic symbol highlighting mode starts when the cursors stays on a symbol for a period longer than the value identified by the **ahs-idle-interval** which defaults to 1.0 second. ☝ To turn off the AHS editing mode, move point away from the highlighted area.		
Edit all highlighted symbols in current function	• **C-c C-d e** • **<f12> e**	(**edts-ahs-edit-current-function**)	Once a symbol is highlighted, use this command to start editing all instances of this symbol in the current function. • Activates ahs-edit-mode with edts-current-function range-plugin.
Edit all highlighted symbols in buffer	• **C-c C-d E** • **<f12> E**	(**edts-ahs-edit-buffer**)	Once a symbol is highlighted, use this command to start editing all instances of this symbol in the current buffer. • Activates ahs-edit-mode with ahs-range-whole-buffer range-plugin.
Move to the next highlighted symbol	**<f12> n**	(**ahs-forward**)	Once a symbol is highlighted, move forward to the next highlighted symbol.
Move to the previous highlighted symbol	**<f12> p**	(**ahs-backward**)	Once a symbol is highlighted, move forward to the previous highlighted symbol.
Move to the originally highlighted symbol	**<f12> .**	(**ahs-back-to-start**)	Once a symbol is highlighted, move back to the symbol that was highlighted at the start of that highlight session.
Refactor: replace region by call to function and add a new function	• **C-c C-d r** • **<f12> r**	(**edts-refactor-extract-function** NAME START END)	Refactor the expression(s) in the region as a function. • The expressions are replaced with a call to the new function, and the function itself is placed on the kill ring for manual placement. The new function's argument list includes all variables that become free during refactoring - that is, the local variables needed from the original function. • New bindings created by the refactored expressions are *not* exported back to the original function. Thus this is not a "pure" refactoring. • This command requires **Erlang syntax_tools** package to be available in the node, version 1.2 (or perhaps later.)
EDTS/Man	EDTS supports opening documentation for a specific function using the information extracted from Erlang Man pages. EDTS maintains a set of Erlang man pages per project, so it is possible to have several Erlang projects each one with a different version of Erlang and their corresponding man pages. These EDTS commands complement the Emacs standard man commands described above in this table.		
Download, install, select Erlang Man pages	**<f12> `**	(**edts-man-setup**)	Download and install OTP man-pages that will be used by the following 2 EDTS commands.
Display help for function at point	• **C-c C-d h** • **<f12> h**	(**edts-show-doc-under-point**)	Find and display the man-page documentation for function under point in a tooltip.
Find and show man-page info for an Erlang module:function	• **C-c C-d H** • **<f12> H**	(**edts-find-doc**)	Prompts for a module, then a function. Find and show the man-page documentation for the Erlang module:function.
EDTS Code Analysis			
Compile current buffer	**<f12> a c**	(**edts-code-compile-and-display**)	Compiles current buffer on node related to that buffer's project.
Run eunit tests	• **C-c C-d t** • **<f12> a t**	(**edts-code-eunit** &optional COMPILATION-RESULT)	Runs eunit tests for current buffer on node related to that buffer's project.
Run dialyzer	**<f12> a a**	(**edts-dialyzer-analyze**)	Runs dialyzer for all live buffers related to current buffer either by belonging to the same project or, if current buffer does not belong to any project, being in the same directory as the current buffer's file.
EDTS/Debug			
Toggle breakpoint	• **C-c C-d b** • **<f12> d b**	(**edts-debug-toggle-breakpoint**)	Toggle breakpoint on current line.
List breakpoints	**C-c C-d M-b** • **<f12> d B**	(**edts-debug-list-breakpoints** &optional SHOW)	Show a listing of all breakpoint on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display process list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
List Erlang processes	• **C-c C-d M-p** • **<f12> d p**	(**edts-debug-list-processes** &optional SHOW)	Show a listing of all processes on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display process list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
Toggle interpretation state of module	• **C-c C-d i** • **<f12> d i**	(**edts-debug-toggle-interpreted**)	Toggle the interpretation state for module in current buffer.
List interpreted modules	• **C-c C-d M-i** • **<f12> d I**	(**edts-debug-list-interpreted** &optional SHOW)	Show a listing of all interpreted modules on all nodes registered with EDTS. If optional argument SHOW is nil or omitted, don't display interpreted list buffer. If it is pop call 'pop-to-buffer', if it is switch call 'switch-to-buffer'.
EDTS/Erlang Node			

Description	Keystroke	Function	Note
Display EDTS Erlang Node Name	**<f12> N**	(**edts-buffer-node-name**)	Print the node sname of the erlang node connected to current buffer. • The node is either: • The module's project node, if current buffer is an erlang module, or • The buffer's erlang node if buffer is an edts-shell buffer. • The project-node of the buffer that was current buffer before jumping to the current buffer if the file of the current buffer is located outside any project (eg. an "externally" loaded module such as an otp-module or a module loaded by ~/.erlang).
Start an EDTS controlled Erlang Shell	**<f12> x**	(**edts-shell** &optional PWD SWITCH-TO)	Start an interactive erlang shell.
Start EDTS server	**<f12> X**	(**edts-api-start-server**)	Starts an edts server-node in a comint-buffer (if not already running).
LSP support: • **lsp-mode** • **erlang_ls**	**LSP (language Server Protocol)** support for Erlang is provided via: • 📦 The **lsp-mode** Emacs Lisp external package PEL activates it when the **pel-use-erlang-ls** user-option is turned on (set to **t**). • The **erlang_ls** Erlang server for LSP. You must install this manually. You will need Git, Erlang, rebar3 and make. The instructions are on the web-site. • 🎛 The **erlang_ls** can be configured using a YAML file **erlang_ls.config** file that must be placed at the root of the Erlang project. ☝ It's important for most projects to set that up, otherwise you may not be able to take advantage of several of the cross-reference features 🚧☝ Both lsp-mode and erlang_ls are under heavy development in the end of 2021. You may want to update these projects regularly to take advantage of the new features and fixes. With PEL you can easily upgrade lsp-mode by moving the its package directory tree into the attic directory and restarting Emacs. If the new version fails just restore the previous one. See the **PEL Manual section on manual package update** for more information.		
• **erlang_ls required environment**	The following executable must be accessible from PATH: • erl, escript and other Erlang executables. See **Installing Erlang** if you need to learn how to install Erlang and its tools. • erlang_ls. To install erlang_ls follow the instruction on the erlang_ls GitHub page: git clone it, then run make and make install. • and the various **Tools for Erlang**.		
• **∑ Customize** lsp-mode	🎛 Several lsp-mode settings are customizable in the **lsp-mode** customization group. With PEL you can access it via **<f12> L <f3>** . The following settings are probably what you may want to customize: • **lsp-log-io** : control whether the LSP process is logging its I/O. Useful for debugging LSP support. • **lsp-ui-sideline-enable** : control whether LSP display information about the current code line. • **lsp-ui-doc-enable** : control whether LSP display documentation about the current code symbol. You can also use the PEL commands to modify them dynamically using the following commands.		
† Toggle code documentation display	**<f11> SCP e L D** **<f12> L D**	(**pel-toggle-lsp-ui-doc** &optional LOCALLY)	Toggle the display of code documentation. • The initial state is set by the '**lsp-ui-doc-enable**' user-option. • By default this command impact is global unless an argument prefix is specified, in which case it is applied to the current buffer only.
† Toggle logging on the LSP client side	**<f11> SCP e L I** **<f12> L I**	(**pel-toggle-lsp-log-io** &optional LOCALLY)	Toggle the logging of LSP I/O. • The initial state is set by the '**lsp-log-io**' user-option. • By default this command impact is global unless an argument prefix is specified, in which case it is applied to the current buffer only. • Once client-side logging is active you can then follow it with lsp-workspace-show-log
† Toggle display of information on current line	**<f11> SCP e L L** **<f12> L L**	(**pel-toggle-lsp-ui-sideline** &optional LOCALLY)	Toggle the display of information of the current line. • The initial state is set by the 'lsp-ui-sideline-enable' user-option. • By default this command impact is global unless an argument prefix is specified, in which case it is applied to the current buffer only.
• **Erlang LS Features**	Overview of the features provided by erlang_ls to LSP-aware editors: • **Breadcrumbs** • **Code completion** • **Go to Definition** • **Go to Implementation of OTP Behaviours** • **Signature Suggestions** • **Diagnostics on file open/save:** • **Compiler Diagnostics** • **Dialyzer Diagnostics** • **Elvis Diagnostics** • **Edoc support** • **Navigation to Included Files** • **Find/Peek References** • **Outline of Module** • **Workspace Symbols** • **Code Folding** • **Insert Code Snippets** • **Suggest Type Specs** • **Automatic Code reloading** • LSP Lenses : lsp-avy-lens • LSP sideline: • enable with: (setq lsp-ui-sideline-enable t) • Use M-x lsp-execute-copde-action to trigger quick-fix actions 🎛 **Erlang Project-Specific LS Configuration**: • Erlang LS is customizable by using a YAML syntax file called **erlang_ls.config** that should be placed in the root directory of the project.		
lsp-mode features	• **Completion at point** • traditional popup with company-mode • **Code navigation**, with • lsp-find-definition • lsp-find-references • **Symbol highlights** • **Code action on mode line** : set **lsp-modeline-code-action-segments** user-option. • **Breadcrumb on headerline**: • Use the **lsp-headerline-breadcrumb-mode** command to toggle their display. The **lsp-headerline-breadcrumb-segments** user-option control what it displays. • **Code Lenses** . The Erlang LS configuration provides • **ct-run-test**: display a *run* button next to a Common Test testcase. • **server-info**: display some Erlang LS server info on top of each module. For debug only. • **show-behaviour-usages**: show the number of modules implementing a behaviour.		
lsp-mode integrations see also: • **∑ Completion/Input** • **∑𝔵 Treemacs** • **∑ Hide/Show**	lsp-mode supports integration with: • 📦 **Helm** by using **helm-lsp** PEL activates when **pel-use-helm-lsp** is turned on. • 📦 **Ivy** by using **lsp-ivy** PEL activates when **pel-use-lsp-ivy** is turned on. • 📦 **treemacs** by using **lsp-treemacs** PEL activates when **pel-use-lsp-treemacs** is turned on. • 📦 **origami** by using **lsp-origami** PEL activates when **pel-use-lsp-origami** is turned on.		
LSP key bindings: • **lsp-mode** • **erlang_ls** See also: **∑ Input Method**	Key bindings: The lsp-mode is a minor mode and provides customizable prefix key for its key bindings. The default key prefix is **s-l**. • Since the **super modifier key** is not always available, it can be modified through customization: change the **lsp-keymap-prefix** value. This can be done with **M-x customize-option** or with PEL via the **<f11> <f2> o** key sequence. • With PEL, the following keys are good replacement candidates: **<f9>** and **C-l** . If you use **<f9>** for Greek letters then consider using **<M-f9>**. • The key bindings shown below show the standard **s-l** key prefix. • If you change **lsp-keymap-prefix** that would be replaced with your selected prefix key.		
LSP Session Control	The Language Server supports several programming languages via their back end. Erlang is supported by the erlang_ls server. • The following commands control the LSP session. • To start a session use the **lsp** command. To stop a session and the server use **lsp-workspace-shutdown**. Use **lsp** to restart a new session. • While a session is running you can restart a session with **lsp-workspace-restart**		
Activate LSP	**s-l w s**	(**lsp** &optional ARG)	Entry point for the server startup. • Without argument: start server if not already running, otherwise display the name:port of the Language Server connected. • With C-u, prompt the user to select which language server to start.
Disconnect LSP	**s-l w D**	(**lsp-disconnect**)	Disconnect the buffer from the language server.
Shut LSP workspace down	**s-l w q**	(**lsp-workspace-shutdown** WORKSPACE)	Shut the workspace WORKSPACE and the language server associated with it. • It may report some errors. The LSP modelling will show disconnected.
Restart the language server	**s-l w r**	(**lsp-workspace-restart** WORKSPACE)	Restart the workspace WORKSPACE and the language server associated with it
LSP Diagnostics	Use the following commands to get information about the currently running Language Server session. • See **Erlang LS Troubleshooting** for more information.		
Describe LSP session	**s-l w d**	(**lsp-describe-session**)	Describes current 'lsp-session'. • Show available tools and the available capabilities • Shows the information inside a LspBrowser buffer.

Description	Keystroke	Function	Note
Validate LSP performance settings	`s-l d`	(**lsp-doctor**)	Validate performance settings and write report in a *lsp-performance* buffer. • It reports as *errors* non-optimal settings. These are not really errors, it's just outlining conditions that are not optimum to get the best performance out of the Language Server.
Toggle LSP protocol logging	`s-l T L`	(**lsp-toggle-trace-io**)	Toggle client-server protocol logging.
Display LSP workspace log buffer	`s-l L`	(**lsp-workspace-show-log** WORKSPACE)	Display the log buffer of WORKSPACE when IO logging is enabled. • With PEL toggle IO logging with `<f12> L D`. • This shows all LSP JSON transactions occurring. ☝ To always see the tail of the buffer move point to the end of buffer first, then leave it there. Emacs will automatically scroll the buffer content to keep the point visible.
Project Setup			
Add directory to the list of workspace folders	`s-l F a`	(**lsp-workspace-folders-add** PROJECT-ROOT)	Add PROJECT-ROOT to the list of workspace folders. • Prompts for the directory.
Remove a directory from the workspace blacklist	`s-l F b`	(**lsp-workspace-blacklist-remove** PROJECT-ROOT)	Remove PROJECT-ROOT from the workspace blacklist.
Remove directory from the list of workspace folders	`s-l F r`	(**lsp-workspace-folders-remove** PROJECT-ROOT)	Remove PROJECT-ROOT from the list of workspace folders.
Toggling features			
Toggle diagnostic modeline	`s-l T D`	(**lsp-modeline-diagnostics-mode** &optional ARG)	Toggle diagnostics modeline.
Toggle current-line status information	`s-l T S`	(**lsp-ui-sideline-mode** &optional ARG)	Minor mode for showing status information for current line. • Displays code status such as definition errors, etc…
Toggle code action on modelling	`s-l T a`	(**lsp-modeline-code-actions-mode** &optional ARG)	Toggle code actions on modeline.
Toggle headline breadcrumbs	`s-l T b`	(**lsp-headerline-breadcrumb-mode** &optional ARG)	Toggle breadcrumb on headerline. • When active the list of directories are listed on the header line. In graphics mode these are buttons you can use to change directory.
Toggle hover information	`s-l T d`	(**lsp-ui-doc-mode** &optional ARG)	Minor mode for showing hover information in child frame. • When active, information about symbol at point is shown in a pop-up overlay area. In graphics mode the information has links that can be used to open web-located information. • For small window the information may cover too much code, use this command to toggle in and out of view. Also note that when the point is toward the bottom of a window the information window may not show completely and you may have to scroll your window.
Toggle symbol highlighting	`s-l T h`	(**lsp-toggle-symbol-highlight**)	Toggle symbol highlighting.
Toggle code-lens	`s-l T l`	(**lsp-lens-mode** &optional ARG)	Toggle code-lens overlays. • Code-lens show information like # times a specific function is referenced. • This can be used to **infer type specs when writing a function**. • For this to work **Dialyzer must be setup** . ⚠ Code lens are overlays and appear above the corresponding line by default. 🐞 There seems to be a **bug in lsp-mode** that prevents scrolling when the overlay hit the top of the window. A work-around is to customize **lsp-lens-place-position** to 'end-of-line instead.
Code Changes			
Reformat Erlang file	`s-l = =`	(**lsp-format-buffer**)	Reformat the code in the current Erlang buffer.
Refactor source import	`s-l r o`	(**lsp-organize-imports**)	Perform the source.organizeImports code action, if available.
Rename symbol at point See also: **∑ Search/Replace**	`s-l r r`	(**lsp-rename** NEWNAME)	Rename the symbol (and all references to it) under point to NEWNAME. ☝For renaming the arguments of a function, the **iedit mode** is more appropriate. It supports restricting the scope to the current function. See **∑ Search/Replace**
Cross Reference			
Find Identifier definitions	`s-l G g`	(**lsp-ui-peek-find-definitions** &optional EXTRA)	Find definitions to the IDENTIFIER at point.
Find symbol implementation locations	`s-l G i`	(**lsp-ui-peek-find-implementation** &optional EXTRA)	Find implementation locations of the symbol at point.
Find references	`s-l G r`	(**lsp-ui-peek-find-references** &optional INCLUDE-DECLARATION EXTRA)	Find references to the IDENTIFIER at point.
Find symbols	`s-l G s`	(**lsp-ui-peek-find-workspace-symbol** PATTERN &optional EXTRA)	Find symbols in the worskpace. The symbols are found matching PATTERN.
Execute code action	`s-l a a`	(**lsp-execute-code-action** INPUT0)	Execute code action ACTION. • If ACTION is not set it will be selected from 'lsp-code-actions-at-point'. • Request codeAction/resolve for more info if server supports.
Highlight all relevant references to symbol at point	`s-l a h`	(**lsp-document-highlight**)	Highlight all relevant references to the symbol under point.
Click LSP lens via avy	`s-l a l`	(**lsp-avy-lens**)	Click lsp lens using 'avy' package. • The code lens must be active. Use `s-l T l` to activate it if it's not active.
Apropos search for symbol/regexp	`s-l g a`	(**xref-find-apropos** PATTERN)	Find all meaningful symbols that match PATTERN. • Can be used to search symbol outside project. • The argument has the same meaning as in 'apropos'. • The result is shown in a *xref* buffer.
Find definitions of symbol at point	`s-l g g`	(**lsp-find-definition** &key DISPLAY-ACTION)	Find definitions of the symbol under point.
Find implementations of symbol at point	`s-l g i`	(**lsp-find-implementation** &key DISPLAY-ACTION)	Find implementations of the symbol under point.

Description	Keystroke	Function	Note
Find references of symbol at point	`s-l g r`	(**lsp-find-references** &optional INCLUDE-DECLARATION &key DISPLAY-ACTION	Find references of the symbol under point. • The result is shown in a *xref* buffer.
Trigger display hover information	`s-l h g`	(**lsp-ui-doc-glance**)	Trigger display hover information popup and hide it on next typing.
Display documentation of symbol at point in *lsp-help*	`s-l h h`	(**lsp-describe-thing-at-point**)	Display the type signature and documentation of the thing at point. • Display help about symbol at point inside a *lsp-help* buffer. ☝ Useful in terminal mode as you can navigate inside the buffer and used other functions to open identified URL references.
Treemacs support • **Treemacs**	📦 The **treemacs** and **lsp-treemacs** external packages activated by PEL user-options **pel-use-treemacs** and **pel-use-lsp-treemacs**, provide extra features that help Erlang development. When these are activated PEL provides bindings for the **lsp-treemacs** features. 🎛 Configure lsp-treemacs by accessing the lsp-treemacs customization group. With PEL use `<f12> w <f3>` from an Erlang buffer.		
• **Open LSP Treemacs error list window.**	`<f12> w e`	(**lsp-treemacs-errors-list**)	Display an error list window at the bottom of the frame. • The buffer uses the treemacs-mode and supports its commands and key bindings. • See **Treemacs** for the list of commands and key bindings. • To close the window, kill its buffer with `C-x k`
• **Quick fix**	`x`	(**lsp-treemacs-quick-fix** &rest ARGS)	If possible, proposes a quick code fix for the error at point.
• **Open LSP Treemacs symbol window**	`<f12> w s`	(**lsp-treemacs-symbols**)	Show symbols view. • To close the window, kill its buffer with `C-x k`
• **Open LSP Treemacs references window**	`<f12> w x`	(**lsp-treemacs-references** ARG)	Show the references for the symbol at point. Issue from an Erlang buffer. • With a prefix argument, select the new window and expand the tree of references automatically. • To close the window, kill its buffer with `C-x k`
• **Open LSP Treemacs implementations window**	`<f12> w i`	(**lsp-treemacs-implementations** ARG)	Show the implementations for the symbol at point. Issue this command from an Erlang buffer. • With a prefix argument, select the new window expand the tree of implementations automatically. • To close the window, kill its buffer with `C-x k`
• **Open LSP Treemacs call hierarchy window**	`<f12> w c`	(**lsp-treemacs-call-hierarchy** OUTGOING)	Show the incoming call hierarchy for the symbol at point. • With a prefix argument, show the outgoing call hierarchy. 🚧 This does not seem to have been implemented for Erlang.
• **Open LSP Treemacs type hierarchy window**	`<f12> w t`	(**lsp-treemacs-type-hierarchy** DIRECTION)	Show the type hierarchy for the symbol at point. • With prefix 0 show sub-types. • With prefix 1 show super-types. • With prefix 2 show both. 🚧 This is not implemented for Erlang.
Rendering markup embedded in comments	The following commands are used to create images from specific markup code embedded inside Erlang 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` `<f11> SCP e u`	(**pel-render-commented-plantuml** PREFIX &optional POS)	Render the PlantUML markup embedded in current mode comment. • Use region if identified otherwise use PlantUML block at point. • Uses prefix (as PREFIX) to choose where to display it: • 4 (when prefixing the command with `C-u`) -> new window • 16 (when prefixing the command with `C-u C-u`) -> new frame. • else -> new buffer • This can be used inside buffer using **any** major mode, when PlantUML markup is embedded inside source code comment.
	☝ Use this in source code to describe your code architecture with PlantUML markup, then generate the UML rendering by moving point inside the PlantUML block and issuing this command. 📦 Requires the **plantuml-mode** external package, activated by **pel-use-plantuml** user option being non-nil.		

Emacs & Erlang— References

Document	Notes
Erlang/OTP	Erlang/OTP home page. This is Erlang's official site.
Erlang versions	• **Erlang Versions - Version Scheme** • **Erlang Support, Compatibility, Deprecations, and Removal**
Erlang/OTP @ Github	Erlang source code
Erlang Community	Links to various topics including how to develop Erlang, learning Erlang, Community mailing lists and chats, contribution, Erlang Issue Tracker, events.
Erlang Mailing Lists	The mailing lists still exist but unfortunately seem to be used less and less.
Erlang/BEAM	Erlang was the first of one of several programming language that runs on the BEAM VM.
Good introduction presentations on Erlang	• **The soul of Erlang and Elixir** • Saša Jurić • GOTO 2019 A very good presentation that captures the essence of why Erlang is so important. Fast pace. A must see. A great presentation to show people that may be reluctant to use the technology. • **The Do's and Don'ts of Error Handling** • **Joe Armstrong** • GOTO 2018

Document	Notes
Erlang References	
Erlang Reference Manual User's Guide	The official Erlang language reference. Lists the BIFs (Built-in functions), reserved words, and all language reference info.
A Concise Guide to Erlang	A very nice quick reference. From David Matuszek, University of Pennsylvania
Erlang Code Guidelines	
Erlang Programming Rules and Conventions	Official Ericsson AB Erlang guidelines.
Inaka's Erlang Coding Standards & Guidelines	Guideline used at Inaka, published on Github.
EDoc User's Guide	Describes how to document code. Comments should conform to the Edoc comment style and format.
2600Hz Erlang Documentation Guideline	An example of a corporate Erlang Documentation Guideline.
Erlang Books	There are several printed and online Erlang books. Erlang's FAQ lists several of them. The following lists some extra ones.
Adopting Erlang	A great and recent (2019 and later) online books on Erlang Development that provides information not available in the Erlang introduction books. Describes how to install Erlang, and how to setup editing tools. A must read to setup Erlang development. This is still work in progress as of May 2020. Each page has a date time stamp.
Erlang Information Sites	
How to setup a local Erlang & Elixir dev environment on Mac from source	LambdaCat post on August 2015. Describes how to use Kerl to install Erlang. Also describes tools to install Elixir. However to get kerl on a macOS machine, using Homebrew is simpler.
• **about-erlang** • **trying-erlang**	These are 2 projects of mine, that I am currently building to centralize some information on Erlang. • **about-erlang** provides general information about Erlang, including: • **Learning Erlang** , a table with links to resources to learn Erlang. • **Installing Erlang**, describes various ways to install Erlang on macOS. • **Tools for Erlang**, describes tools you can use for Erlang development. 🚧
Emacs and Erlang Man files	
How to create a local whatis file	Show how to create a missing whatis file for a set of man pages.
• **The Erlang mode for Emacs (user guide)** • **Erlang mode for Emacs (man page)**	On the erlang.org site. Start here. Describes the 2 files (erlang.el and erlang-start.el) provided by the Erlang mode support, how to set them up for various operating systems. Note, however, that PEL provides the setting for you. It also provides an overview of the various features the package provides. • 🐞 I found bugs in the erlang man page in the **Edit- Moving the marker** section. 1) it's the point that is moved, not the marker, 2) C-a is not an Emacs key prefix, so their key binding descriptions like C-a M-a and C-a M-e are invalid. Reported as ERL-1314. • There's missing information in this. I will identify later as I find out how to get the system going. One aspect to learn more is related to the various erlang-electric functions and variables. • The variable erlang-electric-commands was set to (erlang-electric-comma erlang-electric-semicolon erlang-electric-gt) at first, which does not include the erlang-electric-newline function. I tried adding erlang-electric-newline and activated it, but that made things worse: the newline was no longer automatic after a -> on a function definition line. • Another issue: inside the OS-level erlang shell, we can tab-completion a module:function string, but that does not work inside the emacs erlang shell.
Emacs tools for Erlang	
EDTS	EDTS: stands for: The Erlang Development Tool Suite. See also: • EDTS Tool Suite - Making Your Life Easier - Thomas Järvstrand presentation @ Youtube • EDTS: • configure your project • One Primary EDTS node • 1 node per open project • To setup an Erlang project: a **.edts** file in the project: `:name "my-project"` `:otp-path "path/to/otp"` `:node-name "project-node-name"` `:lib-dirs '("lib" "deps")`
• **How to install EDTS**	Describes some aspects of EDTS and links that may be useful. Lists the requirements. ⚠️🚧After installing EDTS, I got several compile errors, and had to install the following other modules: - auto-complete (v1.5.1) - have to read doc and configure. And perhaps disable company mode?
Language Server Protocol	• Language Server Protocol @ Wikipedia • Language Server Protocol Specifications web site • Language Server Protocol @ Github
• **LSP for Erlang**	LSP support for Erlang is done using the following: • The **lsp-mode** Emacs Lisp package • The **erlang_ls** Erlang server
Erlang Tools accessible via the erlang_ls	Several Erlang tools are available through the erlang_ls LSP server for Erlang. Building erlang_ls from source is the best way to install it as it will also install the secondary tools it provides. • The following is not a complete list of all tools available.
Gradualizer: A Gradual Type System for Erlang	Gradualizer is a type checker for Erlang code based on the principles of **Gradual Typing** that uses the existing Erlang type specs and adds opt-in type checking. It is a work in progress. • Gradualizer @ Github • Youtube presentation: **Dialyzer vs Gradualizer at ElixirConf EU 2019**
company-mode ; Modular in-buffer completion framework for Emacs	
Using Tags with Erlang	
Etags with Erlang @ erlang.org	Describes how to use tags with Erlang source code and how to create the TAGS file.
Troubleshooting	This section describes how to solve some of the problems you may encounter with Erlang on Emacs.
How to prevent Erlang shell echo	On some systems the Erlang shell annoyingly echoes every command typed at the shell. The Emacs manual describes a method to prevent shells inside Emacs from echoing and it describes it as affecting Windows systems. None of the Emacs shells on my system that runs on macOS echo commands, but the Erlang shell does. And the described fix works. PEL activates the fix if the **pel-erlang-shell-prevent-echo** is set to **t**. To activate after setting it: execute pel-init or restart Emacs.