↖ Topic Index

Emacs support for the Erlang Programming Language

Description	Keystroke	Function	Note

Erlang Support
See also:
• **Erlang Reference**
• **Concise Guide To Erlang**
• **about-erlang**
• **Developing Erlang Code with PEL**
  • set PEL Erlang environment
• **∑ Hide/Show**
• **∑ Text Modes**
• **∑ Highlight**
• **∑ Inserting Text**
• **∑ Customize**
☝ ➽ **Identify minor modes to activate automatically in erlang-mode buffers**
• **∑ Speedbar**
	Emacs supports Erlang via the following external packages:
	📦 The **erlang.el** external package (see erlang.el source), part of **OTP** PEL activates it with **pel-use-erlang**.
	📦 The **EDTS** external package PEL activates it with **pel-use-edts** (set to **t** or **start-automatically**).
	📦 The **lsp-mode** external package PEL activates it with **pel-use-erlang-ls**. Uses the **erlang_ls** Erlang LSP server. Integrates with:
	• 📦 **Helm** by using **helm-lsp** PEL activates with **pel-use-helm-lsp**. 📦 **Ivy** by using **lsp-ivy** PEL activates with **pel-use-lsp-ivy**.
	• 📦 **treemacs** by using **lsp-treemacs** PEL activates with **pel-use-treemacs** and **pel-use-lsp-treemacs**.
	• 📦 **origami** by using **lsp-origami** PEL activates with **pel-use-lsp-origami**.
	The **Distel** external package also exists, but seems to have mainly been replaced by EDTS and needs maintenance.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. PEL activates it with **pel-use-hide-comnt**
	📦 The **iedit** external package. PEL activates it with **pel-use-iedit**.
	📦 The **smart-dash** external package. PEL activates it with **pel-use-smart-dash**. erlang-mode is in **pel-modes-activating-smart-dash-mode**.
	📦 The **smartparens** external package. PEL activates it with **pel-use-smartparens**. Add it to **pel-erlang-activates-minor-modes**.
	☛ Useful global minor-modes to activate features in Erlang via **pel-activates-global-minor-mode**: **show-paren-mode**
	🎛 **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 Emacs fill-column value like other major modes.
	  • **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 adds **∑ Speedbar** for .erl, .hrl and .escript Erlang files to show the list of functions.		

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 **Pdf - 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, edts, auto-highlight-symbol, lsp-mode, lsp-ui, lsp-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 the following information in the minibuffer.
	<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	Erlang code syntax highlighting has 4 levels and can be turned off via Erlang menu: **<f10>** to access the menu & select Erlang, then Syntax Highlighting.		
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.
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:
			`sum(L)  -> sum(L, 0).`
			`sum([H|T], Sum) -> sum(T, Sum + H);`
			`sum([], Sum)  -> Sum.`

Description	Keystroke	Function	Note
Electric Keys	The following keys have "*electric*" behaviour and perform special editing tasks to help edit Erlang source code.		
Electric comma	`,`	(**erlang-electric-comma** &optional ARG)	Insert a comma character and possibly a new indented line.
	• The variable 'erlang-electric-comma-criteria' states a criterion, when fulfilled a newline is inserted and the next line is indented. • Behaves just like the normal comma 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.		
Electric semicolon	`;`	(**erlang-electric-semicolon** &optional ARG)	Insert a semicolon character and possibly a prototype for the next line.
	• The variable 'erlang-electric-semicolon-criteria' states a criterion, when fulfilled a newline is inserted, the next line is indented and a prototype for the next line is inserted. Normally the prototype consists of " ->". Should the semicolon end the clause a new clause header is generated. • The variable 'erlang-electric-semicolon-insert-blank-lines' controls the number of blank lines inserted between the current line and new function header. • Behaves just 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.		
Electric > (for the end of arrow)	`>`	(**erlang-electric-gt** &optional ARG)	Insert a greater-than sign, and optionally insert a new line and indent.
Erlang Comments **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 on a code line is controlled by the comment-column variable. Set it with **comment-set-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-l` to indent-for-comment. However PEL uses `M-l` for something else. • The `M-;` binding to comment-dwim works just as indent-for-comment if nothing is marked.		
	`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 See also: ∑ **Comments**	`<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**
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 **fill-column** variable controls where text wraps. • **pel-show-fill-column** `<f11> t f ?` shows its value. Use **set-fill-column** (`C-x f`) to set it. Toggle 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.
Indentation	All syntactic indentation control for Erlang is controlled by the CC-Mode logic and provided commands listed below. • 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**	`<tab>`	(**c-indent-line-or-region** &optional ARG REGION)	Indent active region, current line, or block starting on this line.
	• The indentation level is controlled by the **erlang-indent-level** variable from erlang.el. Its default is 4. • Access its custom group buffer using `<f12> <f3> 1` or `<f11> SPC e <f3> 1`. Or use `<f11> <f2> g erlang RET`. • Note that the erlang.el logic doubles the indentation label inside funs. See this **S.O. discussion on that**. • Behaviour depends on syntactic-indentation mode (enabled by default but can be toggled on/off with the `<f12> M-i` key): • With syntactic-indentation on (the default): • 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. ☝ This might seem strange for new Emacs users, but it ends up being very useful. You can type **<tab>** anywhere in the line to adjust the indentation of the current line or everything in the marked area if a block is marked. • With syntactic-indentation off: • <tab> always indent current line by one level • C-u - <tab> or M- <tab> always un-indent current line by one level • Indenting marked region is done without syntax knowledge and at the same level as previous line. ☝ If you want 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 Erlang function	`C-c C-q`	(**erlang-indent-function**)	Indent current Erlang function. ☝ This also works with a simple tab (see above).
Indent lines of list after point See also: ∑ **Indentation**	`C-M-q`	(**prog-indent-sexp** &optional DEFUN)	Indent the expression after point. When interactively called with prefix, indent the enclosing defun 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.

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>**). However **<f6> p** and **<f6> <up>** handle Shift-marking fine in terminal mode. 🐞Erlang.el man page indicates an invalid mapping for this.
	• **C-M-a** • **C-M-<home>** • **<f6> p** • **<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.
	• **<f6> n** • **<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** 🚧 Experimental support in PEL. Under development. 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 start/end of Blocks**	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_started6() -> ok. ejabberd_started5() -> gen_server:call4(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded3() -> ok. config_reloaded2() -> gen_server:call1(?MODULE, config_reloaded, ?CALL_TIMEOUT).0
• **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()6 -> ok. ejabberd_started()5 -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT)4. -spec config_reloaded()3 -> ok. config_reloaded()2 -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT)1.0
• **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. 0-spec ejabberd_started()1 -> ok. ejabberd_started()2 -> gen_server:call(?MODULE, ejabberd_started, ?CALL_TIMEOUT)3. -spec config_reloaded()4 -> ok. config_reloaded()5 -> gen_server:call(?MODULE, config_reloaded, ?CALL_TIMEOUT)6.
• **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'). 0-spec ejabberd_started1() -> ok. ejabberd_started2() -> gen_server:call3(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -spec config_reloaded4() -> ok. config_reloaded5() -> gen_server:call6(?MODULE, config_reloaded, ?CALL_TIMEOUT).
• **By Blocks and Terms**	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.		
• **Go backward to beginning of previous term/block**	• **C-M-<left>** • **C-[C-b** • **Esc C-b** • **Esc C-<left>** ⚠️ • **C-M-b**	(**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:call9(?MODULE, ejabberd_started, ?CALL_TIMEOUT). -8spec 7config_reloaded6() -> 5ok. 5config_reloaded4() -> 3gen_server:2call1(?MODULE, config_reloaded, ?CALL_TIMEOUT).0 Inside a block: gen_server:call(?3MODULE, 2ejabberd_started, ?1CALL_TIMEOUT0).

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**	(**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 • ∑𝔵 **Smartparens with smartparens-mode active**	**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}}}}.
	• **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, 1907}4}, ☛ 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.
			• The argument INTERACTIVE is for internal use only.
			• 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.
			`music_info() ->`
			`{{er0ror, {noreply, State}}1,` 👈 example
			`{go0od, {{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}.`

	• `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.
			• The argument INTERACTIVE is for internal use only.
			• 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.
			`music_info() ->`
			`6{{error, {noreply, State}},`
			`5{good, 4{{year, 1974},`
			`{group, "Contraction"},`
			`3[{song, "Sam M'Madown"},`
			`{song, "A la claire fontaine"},`
			`{song, "L'alarme à l'oeil"},`
			`2{song, 1"La bourse ou la 0vie"}]`
			`{rating, excellent}}}}.`

Move over space	The commands all use the ∑𝑥 Smartparens external package and required smartparens-mode minor-mode to be active.		

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.
			`start_app(App) ->0 % first clause` 👈 example
			`1start_app(App, temporary).`
			`start_app(App,0 1Type) -> % second clause` 👈 example
			`StartFlag = not is_loaded(),`
			`start_app(App, Type, StartFlag).`

To end of next symbol or block
• **∑𝑥 Smartparens with smartparens-mode active**
	`<M-f7> SPC m`	(**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.
			🚧 This stops inside comments instead of skipping them.
			`start_app(App) -> % first clause` 👈 example
			`start_app(App0 temporary1).`
			`start_app(App0, Type1) -> % second2 clause3` 👈 example 🚧
			`StartFlag4 = not5 is_loaded6(),`
			`start_app7(App8, Type9, StartFlag10).`

To beginning of previous
• **∑𝑥 Smartparens with smartparens-mode active**
	`<M-f7> SPC p`	(**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.
			🚧 This stops inside comments instead of skipping them.
			`10start_app(9App) -> % 8first 7clause` 👈 **Error!** 🚧
			`6start_app(5App, 4temporary).`
			`3start_app(2App, 1Type) ->`
			`0StartFlag = not is_loaded(),` 👈 example
			`start_app(App, Type, StartFlag).`

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.
			`start_app(App) ->0 1% first clause`
			`start_app(App, temporary).`
			`start_app(App, Type) -> % second clause0`
			`1StartFlag = not is_loaded(),`
			`start_app(App, Type, StartFlag).`

Description	Keystroke	Function	Note

Skip backward past whitespace
• ∑ **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.
			`start_app(App) ->1` `0% first clause`
			`  start_app(App, temporary).`
			`start_app(App, Type) ->` `% second clause1`
			`  0StartFlag = not is_loaded(),`
			`  start_app(App, Type, StartFlag).`

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 See also: • ∑ **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 treated as part of words. • With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise. • PEL provides the **<f12> M-p** key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc…)
Marking	The following Erlang-mode specific marking functions are available. They complement what is already available and described in the ∑ **Marking** table. For those 2 commands the Erlang.el man page indicates an invalid mapping for this. Reported as ERL-1314.		
Mark Erlang function	• **C-M-h** • **<f12> f m**	(**mark-defun** &optional ARG) (**erlang-mark-function** &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.
Mark Erlang Clause	• **C-c M-h** • **<f12> c m**	(**erlang-mark-clause**)	Put mark at end of clause, point at beginning.
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.
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.
Enable/Disable coloured highlight of nested blocks (),{},[] See also: ∑ **Highlight**	• **<f12> M-r** • **<M-f12> M-r** • **<f11> h R** • **<f11> SPC e M-r**	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with **M-x customize-group rainbow-delimiters** **Requires:** rainbow-delimiters.el PEL activates this when the **pel-use-rainbow-delimiters** user option is set to **t**.
Inserting code with	**Specialized Tempo Skeletons**		
Insert Parentheses	**M-(**	(**insert-parentheses** &optional ARG)	For Erlang: insert a parenthesis pair '()', leaving point after open-paren. • 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. • 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. • If region is active, insert enclosing characters at region boundaries. • This command assumes point is not in a string or comment.
Insert Erlang Code Templates 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 using the standard tempo skeleton package. • The erlang package make these 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. 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.		
∑ **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.

Description	Keystroke	Function	Note
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. • All formatting is controlled by user-options described above. • 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, you can type the template name (shown in the title column like “if”, “case”, etc) completely or partially 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.
	• **<f11> SPC e <f12> <f12>**		
	• All the tags in the tag lists in ‘tempo-local-tags’ (this includes ‘tempo-tags’) are searched for a match for the text before the point. The way the string to match for is determined can be altered with 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**	(**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.
	• **<f11> SPC e <f12> SPC** • **<f6> SPC**		
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.
Specialized delete and 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> i (** 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.		

Delete content of next block
 	
• ∑𝑥 **Smartparens**
	<M-f7> C-	(**sp-change-inner**)	Change the content of the next block. Point can be anywhere in the 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,{|}};`

Un-wrap current block, splicing its elements in enclosing block	**<M-f7> l l**	(**sp-splice-sexp** &optional ARG)	Un-wrap current block, splicing its content in enclosing block (if any).
			Before:
			`{|EncBytes,EncLen} = 'enc'(Cdx, []),`
			After:
			`|EncBytes,EncLen = 'enc'(Cdx, []),`

Description	Keystroke	Function	Note

• ∑𝑥 **Smartparens**			**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"},}]).

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,|})`

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 whole line	**<M-f7> - l**	(**sp-kill-whole-line**)	⚠️ Currently this deletes the whole line. Requires Erlang specific implementation. 🚧
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 📦external package **flycheck**. • 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-goflymake** 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.

Description	Keystroke	Function	Note
Move to next compile error	• **C-x `** • **M-g n** • **M-g M-n**	(**next-error** &optional ARG RESET)	A prefix ARG specifies how many error messages to move; • negative means move back to previous error messages. • Just **C-u** as a prefix means reparse the error message buffer and start at the first error. ⚠ 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.
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.
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.		

Description	Keystroke	Function	Note
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 See also: • ⅀ 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…
EDTS **Erlang Project settings** See also: ⅀ Sessions	**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**. 👀 EDTS is customizable through it **edts** customization group. With PEL you can open it, with other Erlang specific groups with <f12> <f3>. 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-SPC to activate it. Once it's activated the other keys are available.		
Toggle EDTS mode	<f12> M-SPC <f11> SPC e M-SPC	(**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/Cross References	EDTS provides the following cross-reference commands. It supports navigating in Erlang source code running in the current and remote nodes. ☞ 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.		

Find definition of identifier at point	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.

Go back to where M-. was last issued	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.
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, use a command to 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.		

Description	Keystroke	Function	Note
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			

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).
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.		
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.

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		

• 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 LSP I/O logging	**<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.
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.

Description	Keystroke	Function	Note

• Erlang LS Features	Overview of the features provided by erlang_ls to LSP-aware editors:
	• 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 • Ivy by using lsp-ivy • treemacs by using lsp-treemacs • origami by using lsp-origami	PEL activates when **pel-use-helm-lsp** is turned on. PEL activates when **pel-use-lsp-ivy** is turned on. PEL activates when **pel-use-lsp-treemacs** is turned on. 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.		

Display LSP workspace log buffer	`s-l L`	(**lsp-workspace-show-log** WORKSPACE)	Display the log buffer of WORKSPACE.
Validate LSP performance settings	`s-l d`	(**lsp-doctor**)	Validate performance settings and write report in a *lsp-performance* buffer.
Reformat Erlang file	`s-l = =`	(**lsp-format-buffer**)	Ask the server to format this document.
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.
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.
Toggle diagnostic modeline	`s-l T D`	(**lsp-modeline-diagnostics-mode** &optional ARG)	Toggle diagnostics modeline.
Toggle LSP protocol logging	`s-l T L`	(**lsp-toggle-trace-io**)	Toggle client-server protocol logging.
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.
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.

Description	Keystroke	Function	Note
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.
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.
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**
Disconnect LSP	`s-l w D`	(**lsp-disconnect**)	Disconnect the buffer from the language server.
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.
Shut LSP workspace down	`s-l w q`	(**lsp-workspace-shutdown** WORKSPACE)	Shut the workspace WORKSPACE and the language server associated with it
Restart LSP workspace	`s-l w r`	(**lsp-workspace-restart** WORKSPACE)	Restart the workspace WORKSPACE and the language server associated with it
Activate LSP	`s-l w s`	(**lsp** &optional ARG)	Entry point for the server startup. • When ARG is t the lsp mode will start new language server even if there is language server which can handle current language. • When ARG is nil current file will be opened in multi folder language server if there is such. • When 'lsp' is called with prefix argument ask the user to select which language server to start.
Treemacs support • **∑𝑥 Treemacs**	📦 The **treemacs** and **lsp-treemacs** external packages 🆕 respectively 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 **∑𝑥 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.

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
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.
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
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.