↖ Topic Index

Emacs Support for LFE — Lisp Flavoured Erlang

Description	Keystroke	Function	Note
LFE - Lisp Flavoured Erlang ○ Help & Customization • lfe major mode • useful minor modes ○ Navigation ○ Compile & evaluate • LFE shell • LFE shell commands ○ *LFE reference* Last updated on:	This table describes Emacs support for **LFE** - Lisp Flavoured Erlang - programming language. 📦 LFE support requires the **lfe-mode** external package. 🔄 PEL activates it when the **pel-use-lfe** user-option is turned on (**t**). • Several minor modes can be useful when editing LFE files. They are described below. • File associations: the lfe-mode is activated for files with .lfe, .lfes and .lfesh file extensions. • PEL activates Σ **Speedbar** support for the LFE files when **pel-use-speedbar** user-option is on (set to **t**). 2025-08-26		
Open this PDF file. See also: Σ **Help/Info**	`<f11> SPC C-l <f1>` `<f12> <f1>`	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **LFE** local PDF. If the prefix argument (like **C-u** or **M--**) is used, then it opens the remote GitHub hosted raw PDF instead. If the **pel-flip-help-pdf-arg** user-option is set it's the other way around.
Σ **Customize** PEL LFE support	`<f11> SPC C-l <f2>` `<f12> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL LFE support. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Σ **Customize** Emacs LFE support	`<f11> SPC C-l <f3>` `<f12> <f3>`	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs LFE support: the **lfe** customization group, which controls the settings of the lfe-mode. • If OTHER-WINDOW is non-nil (use **C-u**), display in another window.
Major mode: lfe-mode	The lfe-mode is the major mode used to edit LFE files and buffers.		
Turn lfe-mode on	`M-x lfe-mode`	(**lfe-mode**)	Turn on the lfe-mode, the major mode for LFE buffers. • Automatically activated for LFE files (see the list in the first row above).
Useful Minor Modes to use in lfe-mode	Several minor modes can be used to activate various features when editing LFE files. • PEL provides key binding for toggling several of those minor modes, as shown below. • With PEL, activate a minor mode for LFE files by adding its function name to the **pel-lfe-activates-minor-modes** user-option. • Built-in **show-paren-mode** highlights the parens that matches the one before or after point. • 📦 **Lispy** for short & sweet semantically aware Lisp editing. • 📦 **ParInfer** mode (with either ParInfer Indent Mode or Parinfer Paren Mode) to infer parentheses and indentation. • 📦 **rainbow delimiters** mode, to highlight nested parens with a colour identifying the nesting level. The following commands can be used to toggle useful minor modes for Common Lisp files manually:		
Toggle semantic parser mode on/off	• `<f12> M-s` • `M-<f12> M-s` `<f11> SPC L M-s`	(**semantic-mode** &optional ARG)	Toggle parser features (Semantic mode). • With a prefix argument ARG, enable Semantic mode if ARG is positive, and disable it otherwise. If called from Lisp, enable Semantic mode if ARG is omitted or nil. • In Semantic mode, Emacs parses the buffers you visit for their semantic content.
Toggle Lispy mode See also: **LFE - Lispy**	`<f12> M-L` `<f11> SPC L M-L`	(**pel-lispy-mode** &optional ARG)	Toggle lispy-mode on/off. Lispy is a minor mode for navigating and editing LISP dialects. 📦 Requires lispy external package. 🔄 PEL downloads, installs and configure it when **pel-use-lispy** user option is set to **t**. Please read the information on lispy web site. 💻 **pel-lispy-mode** calls **lispy-mode** but also prepares hydra, loaded dynamically in PEL.
Toggle show-paren mode on/off See also: Σ **Highlight**	• `<f12> M-9` • `M-<f12> M-9` • `<f11> SPC L M-9` • `<f11> h (`	(**show-paren-mode** &optional ARG)	Toggle visualization of matching parens (Show Paren mode). • With a prefix argument ARG, enable Show Paren mode if ARG is positive, and disable it otherwise. • Show Paren mode is a global minor mode. When enabled, any matching parenthesis is highlighted in 'show-paren-style' after 'show-paren-delay' seconds of Emacs idle time.
Enable/Disable coloured highlight of nested blocks (),{},[] See also: Σ **Highlight**	• `<f12> M-r` • `M-<f12> M-r` • `<f11> SPC L m R` • `<f11> h R`	(**rainbow-delimiters-mode** &optional ARG)	Highlight nested parentheses, brackets, and braces with different colours according to their depth. • Customize the depth and colours with `M-x customize-group rainbow-delimiters` 📦 **Requires:** rainbow-delimiters.el 🔄 PEL activates this when the **pel-use-rainbow-delimiters** user option is set to **t**.
Toggle ParInfer mode on/off	• `<f12> M-i` • `M-<f12> M-i` `<f11> SPC L M-i`	(**parinfer-mode** &optional ARG)	Toggle use of the ParInfer mode. In this mode parenthesis depth or indentation is automatically inferred. ⚠ Current implementation of ParInfer does not support hard tabs for indentation. It untabifies and replace them by spaces. ☛ 📦 Requires the parinfer package. ⚠ This is an obsolete package. ☛ 🔄 PEL activates this when the **pel-use-parinfer** user option is set to **t**.
Toggle between ParInfer Indent Mode and Paren Mode	• `<f12> M-I` • `M-<f12> M-I` `<f11> SPC L M-I`	(**parinfer-toggle-mode**)	Switch ParInfer mode between Indent Mode and Paren Mode. ☛ 📦 Requires the parinfer package. ⚠ This is an obsolete package. ☛ 🔄 PEL activates this when the **pel-use-parinfer** user option is set to **t**.
			• ⚠ Note that if the ParInfer mode is not active yet, and it enters ParInfer Indent Mode, the function checks the style of the current buffer and proceed with changing the format after prompting when it finds code that does not conform to the promoted style. The 2 ParInfer modes are: 1. ParInfer **Indent Mode**: • Gives full control of indentation, while ParInfer corrects parens. • Disables the rainbow-delimiter-mode if used, to show closing parens in light gray since they can change as code indentation is changed. • ⚠ When changing to Indent Mode, ParInfer may correct the parentheses format if the code does not corresponds to the promoted style. 2. ParInfer **Paren Mode**: • Gives full control of parens, while ParInfer controls indentation. • Activates rainbow-delimiters-mode if available, showing matching parens in same colors. • 💡 Paren Mode can be used to fix incorrectly indented code before using Indent Mode.
Toggle superword-mode See also: • Σ **Text Modes** • Σ **Search/Replace**	• `<f11> t m p` • `<f12> M-p`	(**superword-mode** &optional ARG)	Toggle superword-mode: a minor mode that treats snake_case as one word. In CommonLisp '-' and '_' are treated as part of words. • With a prefix argument ARG, enable superword mode if ARG is positive, and disable it otherwise. • PEL provides the `<f12> M-p` key for the programming language modes where snake_case is popular (Emacs Lisp, C, C++, Erlang, Python, etc…)
LFE buffer Commands	The lfe-mode behaves like a lisp-mode with the addition of the following commands. We editing a LFE buffer, you can use the lispy-mode which enhance lisp-code editing. See **LFE - Lispy**.		
Insert brackets pair	• `M-[` • `<f12> [` • `M-<f12> [`	(**lfe-insert-brackets** &optional ARG)	Insert a bracket pair. • With numeric argument, enclose as many S-expressions in brackets. • Leave point after open-bracket.
			⚠ The `M-[` key is is the *only* key binding in the lfe-mode key map. But it is a problematic one when Emacs runs in terminal mode because all function keys starting at F5 are encoded with a sequence that begins with the `Esc [` characters. When Emacs is running in terminal mode `M-[` is undistinguishable from `Esc [` . The end result is that all key prefixes using functions keys starting with F5 no longer work properly! To solve that, PEL does the following: • PEL unbinds the `M-[` key when Emacs runs in terminal mode. • PEL adds the `<f12> [` key binding in both terminal and graphics mode.

Description	Keystroke	Function	Note
Navigation in LFE code	This current list below describe the specialized commands only. See the others inside **Navigation** You can also use the lispy mode for extra single key commands for navigation across Lisp source code. See **LFE- Lispy** Several emacs lisp specific commands will also work in a LFE buffer.		

• **To next/previous top-level forms**	Move to beginning /end of S-expression forms. Jump over comments. Can be defun, defer, defconst, defmacros, free-from S-exp, etc…
	The following `**beginning-of-defun**' and `**end-of-defun**' are standard Emacs commands. They have limitations:
	• They only navigate across any top-level form.
	  • They do not discriminate between a **defun**, a **defmacro** or even an **unless** form or any other top-level form.
	  • They do not skip doc-strings unless you set **open-paren-in-column-0-is-defun-start** user option to ignore '(' in strings.
	• PEL provides an additional commands, complementing the standard Emacs commands:
	  • **pel-beginning-of-next-defun** which moves forward to the beginning of the next form
	  • **pel-end-of-previous-defun** which moves backward to the end of the previous top-level form		

Change defun navigation functions (toggle between Emacs default and PEL's)	• **<f12> M-N**
	• **M-<f12> M-N**
	<f11> SPC l M-N
		(**pel-toggle-paren-in-column-0-is-defun-start**)	Toggle interpretation of a paren in column 0 and display new behaviour.
			• It toggles standard Emacs `**open-paren-in-column-0-is-defun-start**' user option, between:
			  • Interpret '(' in column 0 as always stating a defun (even in strings) - the default.
			  • Ignore '(' in strings. A '(' in column 0 is not automatically interpreted as starting a defun.

Backward to beginning of defun See also: **Navigation**	• **C-M-a** • **C-M-<home>** • **<f6> <up>**	(**beginning-of-defun** &optional ARG)	Move backward to the beginning of a top-level form: function definition, macros, etc… • With ARG, do it that many times. Negative ARG means move forward to the ARGth following beginning of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (for **C-M-a** and **C-M-<home>**). It's always available for **<f6> <up>** : hold Shift after typing **<f6>**.

	☝ By default Emacs treats all opening parenthesis character in the first column as a defun.
	  • This causes this function to stop at function definition inside strings.
	  • The behaviour can be changed by setting the open-paren-in-column-0-is-defun-start user option to nil.
	  • PEL provides **pel-toggle-paren-in-column-0-is-defun-start** to toggle that user option. You can also change it dynamically with **<f12> M-N**.
	⚠ Moves to beginning of next function of the same nesting level of the current location. It skips the functions and methods that are more deeply nested.		

Forward to end of defun See also: **Navigation**	• **<f12> <right>** • **M-<f12> <right>** • **C-M-e** • **C-M-<end>** • **<f6> <right>**	(**end-of-defun** &optional ARG)	Move forward to next end of defun. With argument, do it that many times. Negative argument -N means move back to Nth preceding end of defun. ☛ Shift marking is available in graphics mode, not in terminal mode (for **C-M-e** and **C-M-<end>**). **<f6> <right>** and **<f12> <right>** support Shift-marking in terminal mode : hold Shift after typing **<f6>** or **<f12>**. ⚠ This command moves to the end of the next **top-level** function or class.
Forward to start of next top-level form	**<f6> <down>**	(**pel-beginning-of-next-defun** &optional SILENT DONT-PUSH_MARK)	Move forward to the beginning of the next top-level form: function definition, macros, etc.. • Beeps if does not find beginning of next function unless SILENT is non-nil. • If the beginning of next function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. Move back to previous position with **M-`** or **<f6><f6>**. ☛ Shift marking is available with **<f6> <down>** : hold Shift after typing **<f6>**.

	☝ This command is generic and for Emacs Lisp, moves to the beginning of the next top-level form.
	  • It also complements what end-of-defun does. It moves forward but to the beginning of the function definition, which is often what users expect.
	☝ By default Emacs treats all opening parenthesis character in the first column as a defun.
	  • This causes this function to stop at function definition inside strings.
	  • The behaviour can be changed by setting the open-paren-in-column-0-is-defun-start user option to nil.
	  • PEL provides **pel-toggle-paren-in-column-0-is-defun-start** to toggle that user option. You can also change it dynamically with **<f12> M-N**.		

Backward to end of previous defun	• **<f12> <left>** • **M-<f12> <left>** **<f6> <left>**	(**pel-end-of-previous-defun** &optional SILENT DONT-PUSH_MARK)	Move backwards to the end of the previous top-level form. • Beeps if does not find end of previous function unless SILENT is non-nil. • If the end of previous function is found, push the start location to the mark ring unless DONT-PUSH_MARK is non-nil. Move back to previous position with **M-`** or **<f6><f6>**. ☛ Shift marking is available.

• **To next/previous selected S-expression form or defun or …**
 	
★★
	Move to beginning /end of specified S-expression forms. Jump over comments and docstrings. Can be defun, defer, defconst, defmacros, free-from S-exp, groups of them, etc…
	☝ PEL provides the following powerful commands: **pel-elisp-beginning-of-next-form** and **pel-elisp-beginning-of-previous-forms**.
	  • Their behaviour depends on the value of the **pel-elisp-target-forms**, **pel-elisp-user-specified-targets** and **pel-elisp-user-specified-targets2** user-options, as well as their corresponding global or buffer-local values if they exist.
	  • The user options give you the ability to select the type of targets. You can either select the standard behaviour (target the top level forms), or use one of the other 7 types of targets. These include moving to top-level defun form, to any defun form, to defun, defmacro, defsubst, defalias, defadvice forms, to include the eieio forms, the variable definition forms or specify you own set of forms (and those can include the require and provide forms).
	  • More information is available in the docstring of these user options.
	  • When your buffer is using the Emacs-Lisp major mode, use the **<f12> <f2>** key sequence to open the relevant customization buffer which will allow you to see and change the persistent or current session settings.
	☝ PEL also provides specialized versions of these commands:
	  • **pel-elisp-beginning-of-next-defun** which moves to the beginning of next defun, **pel-elisp-beginning-of-previous-defun** to the previous defun.
	  • **pel-elisp-to-name-of-next-defun** which moves to the *name* of the next defun, **pel-elisp-to-name-of-previous-defun** to the previous one.
	  • **pel-elisp-to-name-of-next-form** which moves to the *name* of the next form, **pel-elisp-to-name-of-previous-form** to the previous one.		

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

	<f11> SPC l M-n	• Modifies the value of 'pel-elisp-target-forms' user-option only for the current buffer unless the GLOBALLY argument is non-nil, in which case it modifies the behaviour for all buffers. The change in behaviour does not persist across Emacs sessions. • For persistent change, open the customization buffer with **<f12> <f2>** , modify the value of the **pel-elisp-target-forms**, **pel-elisp-user-specified-targets** and **pel-elisp-user-specified-targets2** user-options and save the customize buffer.	
Forward to start of next definition form ★★ **Configurable target:** • **all top-level forms** • **top-level defun** • **all defun** • **all defun, defsubst, defmacros, …** • **all variable definition forms: defvar, defconst, defcustom, defgroup, …** • **etc…**	• **<f12> <down>** • **M-<f12> <down>** **<f11> SPC l <down>**	(**pel-elisp-beginning-of-next-form** &optional N TARGET SILENT DONT-PUSH-MARK)	Move point forward to the beginning of next N top-level form. • The search is controlled by the value of '**pel-elisp-target-forms**' **pel-elisp-user-specified-targets** and **pel-elisp-user-specified-targets2** user options. That value can be changed for the current session, for all buffers or only for the current buffer by the command 'pel-elisp-set-navigate-target-form', bound to **<f12> M-n**. It can also be specified by the TARGET argument: specify one of the symbols valid for '**pel-elisp-target-forms**'.

	• The function skips over forms inside docstrings.
	• If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success.
	• On success, push original position on the mark ring unless DONT-PUSH-MARK is non-nil.
	• Move back to previous position with **M-`**.
	☛ Shift marking is available with **<f12> <down>**
	☝ This command is the most flexible and can be configured to move like the next 2 commands.
	  • It moves forward but to the beginning of the function definition, which is often what users of other editors expect.
	☝ By default Emacs treats all opening parenthesis character in the first column as a defun: these are top-level forms.
	  • You can change the behaviour: for example, to move to next define or any group of top-level or indented definition forms like defsubst, defmacro, defvar, etc..
	  • The behaviour is customizable (use **<f12> <f2>** then select the **pel-sexp-form-navigation** group to access the relevant user-options: **pel-elisp-target-forms**' , '**pel-elisp-user-specified-targets**' and '**pel-elisp-user-specified-targets2**'. The customization can be saved and then become persistent across Emacs sessions.
	  • You can also control the values of these 2 user-options for all buffers or for each buffer separately:
	  • You can change the values of these variables for a specific buffer or all buffers not yet configured by using the **<f12> M-n** command.
	  • It's possible to set up a buffer to use the **<f12> <down>** key sequence to move to the next defun only or any top-level form, or some other selection or s-expression forms.
	  • Or define your own selection in **pel-elisp-user-specified-targets** and '**pel-elisp-user-specified-targets2**' user-options, then activate them only for a buffer with **<f12> M-n 8** key sequence.
	☝ To count & display # selected forms forward: use a large numeric argument to force a failure: the error message shows number of instances found.
	☝ All of these commands push the point in the mark stack: use **M-`** or **<f6><f6>** to move back to where point was before the command was issued.		

Description	Keystroke	Function	Note
Forward to the name of the next form definition	• **<f12> C-<down>** • **M-<f12> C-<down>**	(**pel-elisp-to-name-of-next-form** &optional N)	Move point to the name of next N defun form - at any level. • Skip over forms located inside docstrings. Leave point on the first character of the form name. • Move back to previous position with **M-`** or **<f6><f6>**.
Forward to beginning of next defun form	• **<f12> M-<down>** • **<f12> f n** • **M-<f12> f n** **<f11> SPC l f n**	(**pel-elisp-beginning-of-next-defun** &optional N)	Move point to the name of the next defun form, whether it is top-level or indented. • The function skips over forms inside docstrings. • Move back to previous position with **M-`** or **<f6><f6>**. • 💻 This uses **pel-elisp-beginning-of-next-form** specifying 'defun-forms as target type. ☛ Shift marking is available with **<f12> M-<down>**
Forward to the name of the next defun definition	• **<f12> C-M-<down>** • **M-<f12> C-M-<down>**	(**pel-elisp-to-name-of-next-defun** &optional N)	Move point to the name of next N defun form - at any level. • Skip over forms located inside docstrings and other types of forms. Leave point on first character of defun name. • Move back to previous position with **M-`** or **<f6><f6>**.
Backward to start of previous definition form ★★ **Configurable target:** • **all top-level forms** • **top-level defun** • **all defun** • **all defun, defsubst, defmacros, …** • **all variable definition forms: defvar, defconst, defcustom, defgroup, …** • **etc…**	• **<f12> <up>** • **M-<f12> <up>** **<f11> SPC l <up>**	(**pel-elisp-beginning-of-previous-form** &optional N TARGET SILENT DONT-PUSH-MARK)	Move point backward to the beginning of previous N top-level form. • The search is controlled by the value of '**pel-elisp-target-forms**' user option. That value can be changed for the current session, for all buffers or only for the current buffer by the command 'pel-elisp-set-navigate-target-form', bound to **<f12> M-n**. It can also be specified by the TARGET argument: specify one of the symbols valid for '**pel-elisp-target-forms**'. ☛ Shift marking is available **<f12> <up>**
	• The function skips over forms inside docstrings. If no valid form is found, don't move point, issue an error describing the failure unless SILENT is non-nil, in which case the function returns nil on error and non-nil on success. • Move back to previous position with **M-`** or **<f6><f6>**. ☝ This command is the most flexible and can be configured to move like the next 2 commands. • It moves backward but to the beginning of the function definition, which is often what users of other editors expect. ☝ By default Emacs treats all opening parenthesis character in the first column as a defun: these are top-level forms. • You can change the behaviour: for example, to move to next define or any group of top-level or indented definition forms like defsubst, defmacro, defvar, etc.. • The behaviour is customizable (use **<f12> <f2>** then select the **pel-sexp-form-navigation** group to access the relevant user-options: **pel-elisp-target-forms**', '**pel-elisp-user-specified-targets**' and '**pel-elisp-user-specified-targets2**'. The customization can be saved and then become persistent across Emacs sessions. • You can also control the values of these 2 user-options for all buffers or for each buffer separately: • You can change the values of these variables for a specific buffer or all buffers not yet configured by using the **<f12> M-n** command. • It's possible to set up a buffer to use the **<f12> <up>** key sequence to move to the previous defun only or any top-level form, or some other selection or s-expression forms. • Or define your own selection in **pel-elisp-user-specified-targets** and '**pel-elisp-user-specified-targets2**' user-options, then activate them only for a buffer with **<f12> M-n 8** key sequence. ☝ To count & display # selected forms backward: use a large numeric argument to force a failure: the error message shows # instances found.		
Backward to the name of the previous form definition	• **<f12> C-<up>** • **M-<f12> C-<up>**	(**pel-elisp-to-name-of-previous-form** &optional N)	Move point to the name of previous N defun form - at any level. • Skip over forms located inside docstrings. Leave point on the first character of the form name. • Move back to previous position with **M-`** or **<f6><f6>**.
Backward to beginning of previous defun form	• **<f12> M-<up>** • **<f12> f p** • **M-<f12> f p** • **<f11> SPC l f p**	(**pel-elisp-beginning-of-previous-defun** &optional N)	Move point to the name of the previous defun form, whether it is top-level or indented. • The function skips over forms inside docstrings. • On success, push original position on the mark ring unless DONT-PUSH-MARK is non-nil. • Move back to previous position with **M-`** or **<f6><f6>**. • 💻 Uses **pel-elisp-beginning-of-previous-form** specifying 'defun-forms as target type. ☛ Shift marking is available with **<f12> M-<up>**
Backward to the name of the previous defun definition	• **<f12> C-<M-up>** • **M-<f12> C-<M-up>**	(**pel-elisp-to-name-of-previous-defun** &optional N)	Move point to the name of previous N defun form - at any level. • Skip over forms located inside docstrings and other types of forms. Leave point on first character of defun name. • Move back to previous position with **M-`** or **<f6><f6>**.
• **By S-Expression form**	Move across forms (S-expressions in Lisp).		
• **By List element**	• Move backward to the beginning or forward to the end of a S-expression form		
Backward block/list See also: **Navigation**	**C-M-p**	(**backward-list** &optional ARG)	Move backward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move forward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-p** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Move block backward See also: • **Navigation**	• **C-M-b** • **C-M-<left>** • **C-[C-b** • **Esc C-b** • **Esc C-<left>** ⚠	(**backward-sexp** &optional ARG)	Move backward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move forward across N balanced expressions. This command assumes point is not in a string or comment. • **C-M-b** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<left>** : ☛ Shift marking works with this command. ❖ **C-M-<left>** does not work on Windows, but **H-<left>** works.
	⚠ With PEL: if you want to use **Esc C-<left>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. 🐧 Several Linux distros map **C-M-<left>** to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.		
Forward block/list See also: **Navigation**	**C-M-n**	(**forward-list** &optional ARG)	Move forward across one balanced group of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. • With ARG, do it that many times. • Negative arg -N means move backward across N groups of parentheses. • This command assumes point is not in a string or comment. • **C-M-n** : ☛ Shift marking is available in graphics mode, not in terminal mode.
Move block forward See also: • **Navigation**	• **C-M-f** • **C-M-<right>** • **C-[C-f** • **Esc C-f** • **Esc C-<right>** ⚠	(**forward-sexp** &optional ARG)	Move forward across one balanced expression (sexp). • With ARG, do it that many times. Negative arg -N means move backward across N balanced expressions. This command assumes point is not in a string or comment. • **C-M-f** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<right>** : ☛ Shift marking works with this command. ❖ **C-M-<right>** does not work on Windows, but **H-<right>** does.
	⚠ With PEL: if you want to use **Esc C-<right>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. 🐧 Several Linux distros map **C-M-<right>** to desktop workspace operation. In that case you can either use another key binding or change Linux key binding in Systems->settings->keyboard->shortcuts to prevent it from using that key sequence.		
• **in/out of lists**	• Move in and out of list nested levels.		
Backward Up/outside sexp hierarchy See also: • **Navigation**	• **C-M-u** • **C-M-<up>** • **C-[C-u** • **Esc C-u** • **Esc C-<up>** ⚠	(**backward-up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move backward out of one level of parentheses. • This command will also work on other parentheses-like expressions defined by the current language mode. With ARG, do this that many times. A negative argument means move forward but still to a less deep spot. • ⚠ With PEL: if you want to use **Esc C-<up>** binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. • **C-M-u** : ☛ Shift marking is available in graphics mode, not in terminal mode. • **C-M-<up>** : ☛ Shift marking works with this command. ❖ **C-M-<up>** does not work on Windows, but **H-<up>** does.

Description	Keystroke	Function	Note
Forward Up/outside sexp hierarchy See also: **Navigation**	`C-M-]`	(**up-list** &optional ARG ESCAPE-STRINGS NO-SYNTAX-CROSSING)	Move forward out of one level of parentheses. • This also works on other parentheses-like expressions defined by the current language mode. • With ARG, do this that many times. A negative argument means move backward but still to a less deep spot. • If ESCAPE-STRINGS is non-nil (as it is interactively), move out of enclosing strings as well. • If NO-SYNTAX-CROSSING is non-nil (as it is interactively), prefer to break out of any enclosing string instead of moving to the start of a list broken across multiple strings. On error, location of point is unspecified.
Forward Down/inside sexp/block See also: • **Navigation**	• `C-M-d` • `C-M-<down>` • `C-[ C-d` • `Esc C-d` • `Esc C-<down>` ⚠️	(**down-list** &optional ARG)	Move forward down one level of parentheses. • This also works on other parentheses-like expressions defined by the current language mode. • With ARG, do this that many times. A negative argument means move backward but still go down a level. • This command assumes point is not in a string or comment. • ⚠️ With PEL: if you want to use `Esc C-<down>` binding you must ensure that **pel-windmove-on-esc-cursor** user option is set to nil. • `C-M-d` : ☛ Shift marking is available in graphics mode, not in terminal mode. • `C-M-<down>` : ☛ Shift marking works with this command. ❖ `C-M-<down>` does not work on Windows, but `H-<down>` does.
• **By sentences**	Move to beginning /end of statement of comment sentence. • The variable 'sentence-end' is a regular expression that matches ends of sentences. Useful in comments. In code it moves to the beginning or end of a definition form (defun, defmacro, etc…)		
Move to beginning of sentence or form	`M-a`	(**backward-sentence** &optional ARG)	Move backward to start of sentence. With arg, do it arg times. ☛ Shift marking works with this command.
Move forward to end of sentence or form	`M-e`	(**forward-sentence** &optional ARG)	Move forward to next end of sentence. With argument, repeat. With negative argument, move backward repeatedly to start of sentence. ☛ Shift marking works with this command.
Compile and evaluate	Use the following commands to evaluate LFE source code in the inferior LFE process where you can then use it. Each of these commands take a prefix argument. You can use any prefix argument like `C-u` or `M--`		
Evaluate the complete buffer	• `<f12> M-c` • `M-<f12> M-c`	(**pel-lfe-eval-buffer** &optional AND-GO)	Send the complete buffer to the inferior LFE process. • Start the inferior LFE process if it's not already running. • Switch to the LFE buffer afterwards when AND-GO argument is non-nil.
Evaluate the S-Expression before point	`C-x C-e`	(**lfe-eval-last-sexp** &optional AND-GO)	Send the previous sexp to the inferior LFE process. • 'AND-GO' means switch to the LFE buffer afterwards.
Evaluate the current region	`C-c C-r`	(**lfe-eval-region** START END &optional AND-GO)	Send the current region (from 'START' to 'END') to the inferior LFE process. • 'AND-GO' means switch to the LFE buffer afterwards.

Open a LFE Shell
 	
(Lisp Flavoured Erlang)
	`<f12> z`
	
	`<f11> z r C-l`
		(**run-lfe** CMD)	Run an inferior LFE process, input and output via a buffer '*inferior-lfe*'.
			• If 'CMD' is given, use it to start the shell, otherwise:
			'inferior-lfe-program' 'inferior-lfe-program-options' -env TERM vt100.
			• If the LFE process is already running move point to its buffer window.
			☛ 📦 Requires the lfe-mode package and LFE (Lisp Flavoured Erlang) installed.
			☛ PEL activates this when the **pel-use-lfe** user option is set to **t**.

LFE Shell	The following PEL commands to open the this PDF and LFE customization buffers are available in the LFE shell buffer. 🎛️ With PEL, activate a minor mode for LFE shell by adding its function name to the **pel-inferior-lfe-activates-minor-modes** user-option.		
Open this PDF file. See also: **Help/Info**	`<f12> <f1>`	(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the **PEL - LFE** 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.
Customize PEL LFE support	`<f12> <f2>`	(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL LFE support. • If OTHER-WINDOW is non-nil (use `C-u`), display in another window.
Customize Emacs LFE support	`<f12> <f3>`	(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs LFE support: the **lfe** customization group, which controls the settings of the lfe-mode. • If OTHER-WINDOW is non-nil (use `C-u`), display in another window.
Shell Commands	The following commands are available in the LFE shell.		
• **text input**	The following commands control the input text prepared to send to the LFE process.		
Delete character forward	`C-d`	(**comint-delchar-or-maybe-eof** ARG)	Delete ARG characters forward or send an EOF to subprocess. • Sends an EOF only if point is at the end of the buffer and there is no input.
Delete character backward	`<delete>`	(**backward-delete-char-untabify** ARG &optional KILLP)	Delete characters backward, changing tabs into spaces. • The exact behavior depends on 'backward-delete-char-untabify-method'. • Delete ARG chars, and kill (save in kill ring) if KILLP is non-nil. • Interactively, ARG is the prefix arg (default 1) and KILLP is t if a prefix arg was specified.
Delete current input text	`C-c C-u`	(**comint-kill-input**)	Kill all text from last stuff output by interpreter to point. ☝ Quick way to delete the complete input line after the prompt.
Delete previous word	`C-c C-w`	(**backward-kill-word** ARG)	Kill characters backward until encountering the beginning of a word. • With argument ARG, do this that many times.
Prompt history: next	`C-<down>`	(**comint-next-input** ARG)	Cycle forwards through input history.
Prompt history: previous	`C-<up>`	(**comint-previous-input** ARG)	Cycle backwards through input history, saving input.
Format typed input			
Continue entry on new line without sending to LFE	`C-c SPC`	(**comint-accumulate**)	Accumulate a line to send as input along with more lines. • This inserts a newline so that you can enter more text to be sent along with this line. Use RET to send all the accumulated input, at once. The entire accumulated text becomes one item in the input history when you send it.
Indent all lines of a list starting just after point	`C-M-q`	(**indent-sexp** &optional ENDPOS)	Indent each line of the list starting just after point. • If optional arg ENDPOS is given, indent each line, stopping when ENDPOS is encountered. ☛ In the LFE shell use this with `C-c SPC` to build a list with its elements on multiple lines and indent it. When at the end of the list, terminate the S-expression, return to the top of the input with `C-c C-a` and then type `C-M-q`
Indent the current line of the LFE shell input	`<tab>`	(**indent-for-tab-command** &optional ARG)	Indent the current line or region, or insert a tab, as appropriate. • This function either inserts a tab, or indents the current line, or performs symbol completion, depending on 'tab-always-indent'. The function called to actually indent the line or insert a tab is given by the variable 'indent-line-function'. • If a prefix argument is given, after this function indents the current line or inserts a tab, it also rigidly indents the entire balanced expression which starts at the beginning of the current line, to reflect the current line's indentation. • In most major modes, if point was in the current line's indentation, it is moved to the first non-whitespace character after indenting; otherwise it stays at the same position relative to the text. • If 'transient-mark-mode' is turned on and the region is active, this function instead calls 'indent-region'. In this case, any prefix argument is ignored.

Description	Keystroke	Function	Note
• **send to LFE process**	Use the following commands to send text or signal to the LFE process.		
Send input	**`RET`**	(**comint-send-input** &optional NO-NEWLINE ARTIFICIAL)	Send input to process.
Send EOF	**`C-c C-d`**	(**comint-send-eof**)	Send an EOF to the current buffer's process.
Interrupt/break LFE	**`C-c C-c`**	(**comint-interrupt-subjob**)	Interrupt the current subjob. The LFE process stops and displays the break prompt: `BREAK: (a)bort (A)bort with dump (c)ontinue (p)roc info (i)nfo (l)oaded (v)ersion (k)ill (D)b-tables (d)istribution` ⚠ Depending on the state of the system, this does not always seem to work properly. If the buffer freezes type **`C-g`** to restore control.
Stop current subjob	**`C-c C-z`**	(**comint-stop-subjob**)	Stop the current subjob. ⚠ if there is no current subjob, you can end up suspending the top-level process running in the buffer. If you accidentally do this, use **`M-x comint-continue-subjob`** to resume the process. (This is not a problem with most shells, since they ignore this signal.)
Send quit signal to LFE subjob	**`C-c C-\`**	(**comint-quit-subjob**)	Send quit signal to the current subjob.
Navigate	Use the following commands to navigate across LFE prompts.		
Move point to beginning of line, then to process mark	**`C-c C-a`**	(**comint-bol-or-process-mark**)	Move point to beginning of line (after prompt) or to the process mark. • The first time you use this command, it moves to the beginning of the line (but after the prompt, if any). If you repeat it again immediately, it moves point to the process mark. • The process mark separates the process output, along with input already sent, from input that has not yet been sent. Ordinarily, the process mark is at the beginning of the current input line; but if you have used **`C-c SPC`** to send multiple lines at once, the process mark is at the beginning of the accumulated input.
Move point to next prompt entry	**`C-c C-n`**	(**comint-next-prompt** N)	Move to end of Nth next prompt in the buffer. • If 'comint-use-prompt-regexp' is nil, then this means the beginning of the Nth next 'input' field, otherwise, it means the Nth occurrence of text matching 'comint-prompt-regexp'.
Move point to previous prompt entry	**`C-c C-p`**	(**comint-previous-prompt** N)	Move to end of Nth previous prompt in the buffer. • If 'comint-use-prompt-regexp' is nil, then this means the beginning of the Nth previous 'input' field, otherwise, it means the Nth occurrence of text matching 'comint-prompt-regexp'.
Reposition buffer	Use the following commands to re-position the buffer inside its window, operations that are similar to scrolling.		
Display last output at the top of the window	**`C-c C-r`**	(**comint-show-output**)	Display start of this batch of interpreter output at top of window. • Sets mark to the value of point when this command is run.
Put end of buffer at the end of the window	**`C-c C-e`**	(**comint-show-maximum-output**)	Put the end of the buffer at the bottom of the window.
	`C-c C-m`	(**comint-copy-old-input**)	Insert after prompt old input at point as new input to be edited. • Calls 'comint-get-old-input' to get old input.
Write interpreter output to specified file	**`C-c C-s`**	(**comint-write-output** FILENAME &optional APPEND MUSTBENEW)	Write output from interpreter since last input to FILENAME. • Any prompt at the end of the output is not written. • If the optional argument APPEND (the prefix argument when interactive) is non-nil, the output is appended to the file instead. • If the optional argument MUSTBENEW is non-nil, check for an existing file with the same name. If MUSTBENEW is 'excl', that means to get an error if the file already exists; never overwrite. If MUSTBENEW is neither nil nor 'excl', that means ask for confirmation before overwriting, but do go ahead and overwrite the file if the user confirms. When interactive, MUSTBENEW is nil when appending, and t otherwise.
Cleanup output			
Delete last output	**`C-c C-o`**	(**comint-delete-output**)	Delete all output from interpreter since last input. • Does not delete the prompt.
Clean LFE shell buffer	**`C-c M-o`**	**(inferior-lfe-clear-buffer)**	Delete the output generated by the LFE process. ☛ All lines before the current prompt are deleted from the buffer. The Emacs-maintained history is still available.
Search command history	Use the following commands to retrieve a similar command from the command history		
Display command history in the *Input History* buffer	**`C-c C-l`**	(**comint-dynamic-list-input-ring**)	Display a list of recent inputs entered into the current buffer.
Previous matching history entry	**`C-c M-r`**	(**comint-previous-matching-input-from-input** N)	Search backwards through input history for match for current input. • (Previous history elements are earlier commands.) • With prefix argument N, search for Nth previous match. • If N is negative, search forwards for the -Nth following match.
Next matching history entry	**`C-c M-s`**	(**comint-next-matching-input-from-input** N)	Search forwards through input history for match for current input. • (Following history elements are more recent commands.) • With prefix argument N, search for Nth following match. • If N is negative, search backwards for the -Nth previous match.
Insert nth argument used in call of the previous command	**`C-c .`**	(**comint-insert-previous-argument** INDEX)	Insert the INDEXth argument from the previous Comint command-line at point. • Spaces are added at beginning and/or end of the inserted string if necessary to ensure that it's separated from adjacent arguments. • Interactively, if no prefix argument is given, the last argument is inserted. • Repeated interactive invocations will cycle through the same argument from progressively earlier commands (using the value of INDEX specified with the first command). • This command is like 'M-.' in bash.

LFE — References

Document	Notes
LFE - Lisp Flavored Erlang	
LFE @ Wikipedia	Has a quick overview of the language.
LFE Home page	LFE Home
LFE - Emacs Support	
lfe-mode @ GitHub	LFE Emacs support written by Robert Virding
flycheck-rebar3 @ GitHub	Flycheck integration for rebar3 projects
LFE - Books	**Published LFE Books repository @ GitHub** , a list of published LFE books.
Quick Start with rebar3_lfe	Start by reading this book that presents how to build LFE projects and introduces LFE and rebar3.
The LFE Tutorial	Getting started with LFE
Casting SPELs in LFE	A port of **Casting Spells in Lisp** using LFE. It describes how to use LFE to write distributed, fault-tolerant, message-passing game application.
Structure and Interpretation of Computer Programs - The LFE Edition	A revisit of the MIT classic SICP book using LFE. An in-going project (source at GitHub here) with the first 2 chapters completed as of May 2021.
LFE - References	
LFE Guide	
LFE Style Guide	The LFE Style Guide @ GitHub
Data Types in LFE	
LFE REPL functions, environment, variables, etc…	
LFE compatibility with Common Lisp	
LFE compatibility with Clojure	
LFE and Docker	
LFE - Presentation Videos	
LFE	Robert Virding - LFE - a lisp flavour on the Erlang VM (Lambda Days 2016) - Video presentation @ YouTube where Robert Virding introduces LFE .
LFE & Flavors for LFE	Robert Virding - Lisp Machine Flavors for LFE implementing objects on Erlang OTP - EEF17 Conference @ YouTube
LFE - Code Examples	
lfe / examples @ Github	A set of examples you can use to learn LFE
LFE @ RosettaCode	More LFE code examples identified by categories
LFE - Libraries	
hex.pm - The package manager for Erlang ecosystem	
• **lfe packages registered @ hex.pm**	
• **LFE - Libraries - Flavors**	**Flavors** is on object-oriented extension of Lisp and had a strong influence on the design **CLOS (Common Lisp Object System)**.
LFE Flavors @ GitHub	The LFE implementation of Flavors.
Introduction to Flavors	The 1985 manual, adapted from text written by David Moon and Daniel Weinreb.
Flavors 1.1.1.1 Documentation @ Franz Lisp	The current Flavors manual at Franz Lisp
LFE - Tools	

rebar3 — The official build tool for Erlang - @ rebar3.org
 	
rebar3 @ Github
	The rebar3 tool is required for Erlang and LFE development.
	• **rebar3.org**
	• Links to some of the main sections of the manual:
	  • **Getting started with rebar3 - documentation** - start by reading this.
	  • To install Erlang you may want to read **Installing Erlang** in my **about-erlang project** : it provides several links about multiple ways to install Erlang including the Adopting Erlang link included in the rebar3 manual.
	  • **Basic usage**
	  • **Workflow**
	  • **Configuration**
	  • **Commands**
	  • **Testing**

rebar3_lfe @ GitHub • **LFE rebar3 Plugin manual**	"*A comprehensive LFE rebar3 plugin for all your LFE tooling needs*". • Tool written by Duncan McGreggor • Extends rebar3 with LFE specific commands. • Start by reading its **setup** and **features** • There's a separate **LFE rebar3 Plugin manual**