↖ Topic Index

ERT — Emacs Lisp Regression Testing

Description	Keystroke	Function	Note
Using ERT ○ Help & Customization • Run tests interactively • Byte Compile & run tests • *ert* buffer commands ○ command line tests • Emacs Lisp test code facilities • Test coverage highlighting ○ ERT Reference • ERT Manuals • ERT Tools • Articles on ERT • Other unit testing tools • Emacs Lisp Concepts • CI/CD Test on Github	If you write Emacs Lisp code, writing unit test code normally helps increase code quality and maintainability. For 𝔏𝔓𝔏 - Emacs Lisp ERT is a simple, yet powerful, environment to write test you can run from the command line and interactively inside Emacs with the ability to debug failing code. • ERT is part of Emacs standard distribution since Emacs 24. • ERT provides a command to run tests that have been written using the ERT macros. • Tests are normally written inside separate .el files, with names using the same prefix as the file being tested and also often placed inside a test directory. **To run the test**: 1. Load the Emacs Lisp file that defines the tests. You can also visit and byte compile the file (with PEL: **`<f12> c b`**). • With PEL you can use the **`<f12> l f`** or **`<f12> l v`** key bindings to load-file and **pel-load-visited-file** commands. • See 𝔏𝔓𝔏 - Emacs Lisp for more info. 2. Type **`M-x ert RET t RET`**. • That executes the **ert** command described below and provides the selector **t** which means running every test. • You can also use other regular expression that identify the names of the test functions to run: • for example use "^foo-" to run all tests that have a name that begins with foo-. You can extend ERT testing capabilities with several external packages and libraries. PEL support the automatic installation and setup of the following ones. Just turn on the corresponding PEL user-option to install and activate them.		
• Emacs Lisp language extension.	**noflet** (PEL uses **my fork**)	**pel-use-noflet**	Locally override a function in the manner of **flet** but with access to the original function though the **this-fn** symbol. PEL uses **my fork** that integrates all fixes provided by many.
		☝Unless you need `noflet` **`this-fn`** feature, you can use `cl-lib` `cl-letf` instead of `noflet`.	
• Mocking Libraries	**el-mock**	**pel-use-el-mock**	A tiny Mock and Sti=ub framework for Emacs Lisp.
	el-spy	**pel-use-el-spy**	A small mocking framework for Emacs Lisp that also support spy and proxy.
	mocker	**pel-use-mocker**	A simple mocking framework for Emacs.
• Test coverage highlighting	**coverlay**	**pel-use-coverlay**	Test coverage overlay. Supports **LCOV** format (which **Coveralls** uses).
	coverlage	**pel-use-coverage**	An older test coverage system, which depends on even older ert-expections.
	ert-expectations	**pel-use-ert-expectations**	A nice wrapper around ert to create test simply. However the code is old and would need updates.
	test-cover-mark	**pel-use-testcover-mark**	Mark the whole line of code not completely tested. Extends what builtin **testcover** marks.
• ERT test runner	**ert-runner**	**pel-use-ert-runner**	ERT runner to run ERT test in CI/CD systems. Often used with **Cask** to drive ERT tests.
• Alternative test frameworks	**buttercup**	**pel-use-buttercup**	Behavior-Driven Emacs Lisp Testing. An alternative to ERT that addresses several limitations of ERT.
Last updated on:	2026-01-15	**`<f12>`** ERT commands key bindings refer to kets typed inside an Emacs Lisp buffer. See more info in 𝔏𝔓𝔏 - Emacs Lisp.	

Open this PDF file.
See also: Σ **Help/Info**
	`<f12> C-t <f1>`
	`<f11> SPC l C-t <f1>`
		(**pel-help-pdf** &optional OPEN-WEB-PAGE)	Open the 𝔓𝔏 - **ERT** 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 ERT support	**`<f12> <f2>`**
	`<f11> SPC l C-t <f2>`
		(**pel-customize-pel** &optional OTHER-WINDOW)	Customize PEL ERT support.
			• If OTHER-WINDOW is non-nil (use **`C-u`**), display in another window.

Σ **Customize** Emacs ERT support	**`<f12> <f3>`**
	`<f11> SPC l C-t <f3>`
		(**pel-customize-library** &optional OTHER-WINDOW)	Customize Emacs ERT support: overlay, testcover, testcover-mark-line
			• If OTHER-WINDOW is non-nil (use **`C-u`**), display in another window.

Byte Compile and run all Emacs Lisp tests	**`<f12> C-t C-t`**	(**pel-run-ert**)	Byte compile and run ERT test on current 𝔏𝔓𝔏 - **Emacs Lisp** buffer. • Prompts if the buffer needs to be saved first.
Run test interactively	**`<f12> C-t C-r`**	(**ert** SELECTOR &optional OUTPUT-BUFFER-NAME MESSAGE-FN)	Run the tests specified by SELECTOR and display the results in a buffer. • SELECTOR works as described in 'ert-select-tests'. (Use **t** to run all tests, or name the test to execute.
	• By default, the results are stored inside the *ert* buffer, opened in ERT-Results mode. ⚠️ ert does not save and byte compile the buffer before attempting to run the test. It must be done before. With PEL, use **pel-run-test**. See below.		
Extra 2 args, only available in Emacs < 29	The OUTPUT-BUFFER-NAME and MESSAGE-FN optional arguments were available until Emacs 29. Starting at Emacs 29 they are no longer available. OUTPUT-BUFFER-NAME and MESSAGE-FN should normally be nil; they are used for automated self-tests and specify which buffer to use and how to display message.		
***ert* buffer commands** (Available from the *ert* buffer opened once art runs a test.)	In the *ert* buffer, the result of each test appears as a coloured character button: • A period **.** if the test passed. Coloured in green. • A red upper case **F** if the test failed. • A green lower case **f** if the test failed, but was expected to fail (the **ert-deftest** has a **:expected-result :failed**). Used to identify bugs not yet fixed. • A red upper case **P** indicating a test passed but was expected to fail. • While at a button you can issue other commands. The following single key commands are available when point is inside the buffer's window:		
Move to test	**`TAB`**	(**forward-button** N &optional WRAP DISPLAY-MESSAGE)	Move point to the next test result button.
Re-run test	**`r`**	(**ert-results-rerun-test-at-point**)	Re-run the same test
Jump to test source	**`.`**	(**ert-results-find-test-at-point-other-window**)	Jump to the source code of the test (in another window)
List *should* forms	**`l`**	(**ert-results-pop-to-should-forms-for-test-at-point**)	Shows the list of all *should* forms executed during the test before it failed
View backtrace	**`b`**	(**ert-results-pop-to-backtrace-for-test-at-point**)	View the backtrace for the failed test at point
Re-run test with debugging	**`d`**	(**ert-results-rerun-test-at-point-debugging-errors**)	Re-run the same test with debugging enabled
Show messages	**`m`**	(**ert-results-pop-to-messages-for-test-at-point**)	Show what messages were printed before the test failed
Toggle condition printing	**`L`**	(**ert-results-toggle-printer-limits-for-test-at-point**)	Toggle how much of the condition to print for the test at point.
Delete obsolete tests	**`D`**	(**ert-delete-test** TEST-NAME)	Delete obsolete tests (test whose code might have changed)
Re-run all tests	**`R`**	(**ert-results-rerun-all-tests**)	Re-run all tests, using the same selector
Move to the next test result	**`n`**	(**ert-results-next-test**)	Move to the next test results.
Move to previous test result	**`p`**	(**ert-results-previous-test**)	Move to the previous test results.
Jump between summary and result	**`j`**	(**ert-results-jump-between-summary-and-result**)	Jump between test result and summary.
	By positioning point on the test result character (., F or f) and then typing **j** point will move to the test summary (and will create one if the test passed). From the summary you can easily press RET to move point to the source code of the test (in another window).		
Show help for test	**`h`**	(**ert-results-describe-test-at-point**)	Get help for the test corresponding to the test result character (or test summary) at point.
Describe available commands	**`?`**	(**describe-mode** &optional BUFFER)	Describe mode (and these commands)
Quit - close window	**`q`**	(**quit-window** &optional KILL WINDOW)	Quit window - dismiss the *ert* popup buffer window.

Description	Keystroke	Function	Note
Run Tests in Batch Mode	To execute tests from the command line, use Emacs in batch mode to load the specific tests and run them using one of the following ERT functions. • For example: `emacs -batch -l ert -l my-tests.el -f ert-run-tests-batch-and-exit`		
Run batch test		(**ert-run-tests-batch** &optional SELECTOR)	• Run the tests specified by SELECTOR, printing results to the terminal.
	• SELECTOR works as described in 'ert-select-tests', except if SELECTOR is nil, in which case all tests rather than none will be run; this makes the command line "emacs -batch -l my-tests.el -f ert-run-tests-batch-and-exit" useful. Returns the stats object.		
Run batch tests and exit		(**ert-run-tests-batch-and-exit** &optional SELECTOR)	Like 'ert-run-tests-batch', but exits Emacs when done.
	• The exit status will be 0 if all test results were as expected, 1 on unexpected results, or 2 if the tool detected an error outside of the tests (e.g. invalid SELECTOR or bug in the code that runs the tests).		
Emacs Lisp Test Code	Test code is written using forms that use the ert-deftest macro. See the following macro descriptions		
Define a test		(**ert-deftest** NAME () [DOCSTRING] [:expected-result RESULT-TYPE] [:tags '(TAG...)] BODY...)	Define NAME (a symbol) as a test. • BODY is evaluated as a 'progn' when the test is run. It should signal a condition on failure or just return if the test passes. • '**should**', '**should-not**', '**should-error**' and '**skip-unless**' are useful for assertions in BODY. • Use 'ert' to run tests interactively. • Tests that are expected to fail can be marked as such using :expected-result. See 'ert-test-result-type-p' for a description of valid values for RESULT-TYPE.
The should macro		(**should** FORM)	Evaluate FORM. If it returns nil, abort the current test as failed. • Returns the value of FORM.
The should-not macro		(**should-not** FORM)	Evaluate FORM. If it returns non-nil, abort the current test as failed. • Returns nil.
The should-error macro		(**should-error** FORM &rest KEYS &key TYPE EXCLUDE-SUBTYPES)	Evaluate FORM and check that it signals an error.
	• The error signaled needs to match TYPE. TYPE should be a list of condition names. (It can also be a non-nil symbol, which is equivalent to a singleton list containing that symbol.) If EXCLUDE-SUBTYPES is nil, the error matches TYPE if one of its condition names is an element of TYPE. If EXCLUDE-SUBTYPES is non-nil, the error matches TYPE if it is an element of TYPE. • If the error matches, returns (ERROR-SYMBOL . DATA) from the error. If not, or if no error was signaled, abort the test as failed.		
The skip-unless macro		(**skip-unless** CONDITION)	Skip the current ert-deftest defined test unless CONDITION is non-nil. • Normally used to skip tests when the environment does not provide the necessary conditions for a valid test.
The :expected-result tag		**:expected-result** • **:failed** • **:passed**	Tag tests of lesser importances that are expected to fail potentially under some conditions. It can also be used to silence the report of a bug while leaving in the test. • See the documentation.
Test Coverage highlighting	To measure test coverage of elisp code in a file first activate the coverage and then run the tests that exercise the code. After the test you can then activate marking of areas that have not been tested.		
Identify file to mark code being use during upcoming test.	`<f12> C-t C-c C-s`	(**testcover-start** FILENAME &optional BYTE-COMPILE)	Prompt for FILENAME, then use Edebug to instrument for coverage all macros and functions inside it. • If BYTE-COMPILE is non-nil, byte compile each function after instrumenting.
Start test coverage for current function	`<f12> C-t C-c .`	(**testcover-this-defun**)	Start coverage on function under point.
Overlays all forms that have low test coverage	`<f12> C-t C-c C-a`	(**testcover-mark-all** &optional BUFFER)	Mark all forms in BUFFER that did not get completely tested during coverage tests. • This function creates many overlays in the buffer.
Remove coverage overlays	`<f12> C-t C-c C-u`	(**testcover-unmark-all** BUFFER)	Remove all overlays from FILENAME.
Move point to next line with low coverage.	`<f12> C-t C-c C-n`	(**testcover-next-mark**)	Move point to next line in current buffer that has a splotch.
Toggle global minor mode to mark whole lines	`<f12> C-t C-c C-l`	(**testcover-mark-line-mode** &optional ARG)	Toggle global minor mode to mark whole line with testcover. 📦 Requires **test-cover-mark** activated by **pel-use-testcover-mark**
Extend the test cover overlays to eat entire line	`<f12> C-t C-c C-m`	(**testcover-mark-line-mark-all**)	Mark all forms that didn't get completely tested, with lines, extending the overlays that are created by test cover itself. 📦 Requires **test-cover-mark** activated by **pel-use-testcover-mark**
End coverage instrumentation.	`<f12> C-t C-c C-e`	(**testcover-end** FILENAME)	Turn off instrumentation of all macros and functions in FILENAME.

Emacs Lisp Testing — References

Topic & Link	Description
ERT Manuals	
ERT : Emacs Lisp Regression Testing	ERT Manual, part of Emacs.
Test Coverage — Emacs Lisp	Test coverage section of the GNU Emacs Lisp manual
ERT Tools Repositories and Files	
pel-ert.el	pel-ert.el defines a set of equality predicates that accept extra arguments for the sole purpose of having them shown in the ERT report of a failed test. To see a set of *test environment* variables in the test report just pass them as extra arguments to these equality predicates.
testcover.el source	Source of the test coverage support
overseer.el @ GitHub	
ert-runner @ GitHub	
Emacs Lisp Mock @ EmacsWiki	The original location of that mock library that can be used with ert.
El mock @ GitHub	The new location for el-mock.el
emacs-noflet @ GitHub	External package that provides the noflet macro which can be used to re-define functions locally. Can be used in ERT testing. I created my own fork of this and integrated most long standing pull requests. See pierre-rouleau / emacs-noflet .
Articles/Blogs on ERT	Interesting articles to read before writing Emacs Lisp ERT testing code.
Quick intro to ert testing	A quick overview of ERT based Emacs Lisp testing.
Elisp Unit Testing with ERT	Quick overview of ERT in an August 2012 blog written by Chris Wellons. • Since then the cl.el library was replaced by the cl-lib.el and the flet function was deprecated to cl-flet, but aside from these small items the description is still valid.
flet, cl-flet, cl-letf and noflet	• **Make Flet Great Again :** Another great post from Chris Wellons. • Describes the new cl-flet and cl-letf, and how to use cl-letf to create Ert test that modify the behaviour of called functions. • **Understanding letf and how it replaces flet**, by Arthur Malabarba • Describes cl-letf and how it can replace the old flet.
ERT: Emacs Lisp Regression Testing	A nice description of the ERT features and techniques given at the NYC meetup. Describes: • How to write a simple test, how to run the test and use the *ert* buffer commands. • How to test expected errors with should-error
Continuous integration and code coverage for Emacs packages with Travis and Coveralls	An overview of automated Emacs Lisp unit testing with coverage measurement from Sacha Chua blog.
Reddit discussion: Testing in Emacs	Interesting discussion on testing under Emacs. Leads to several interesting projects: • **test-cockpit** for testing code written in several programming languages under Emacs • **dape** : Debug Adapter Protocol for Emacs • **stan-mode** which uses buttercup for testing.
Other Unit Testing Support	Other packages and libraries for testing are not listed at the top of this page and currently not explicitly supported by PEL. Some are listed here.
• **Unit Testing on EmacsWiki**	Provides a list of alternatives to ERT with other testing-related information.
• **emacs-test-simple @ GitHub**	An Alternative to ERT developed by debugger expert R.Bernstein (see also here).
• **undercover.el**	A test coverage library for Emacs. Does not support byte-compilation nor circular objects. Meant to be used with Cask, Eask or Eldev.
Shell based unit test tools	
cram	A simple test system to test shell commands. cram is a command line tool that takes cram scripts stored in .t files. • PEL supports the **cram-mode** when pel-use-cram-mode is turned on.
Emacs Lisp Concepts	Several Emacs Lisp concepts are useful when writing ERT test and mock ups.
Scoping Rules for Variable Bindings	When writing Emacs Lisp code and test with mockup in particular its important to fully understand the concepts of dynamic and lexical binding in Emacs Lisp. ⚠ Be aware that starting with version 27 of Emacs Lisp lexical binding is the default while dynamic binding was the default in previous versions.
Github Continuous Integration	
• **With Github Actions**	• Workflow syntax for GitHub Actions • Github Actions Runner Images : select the os environment • Setup Emacs for Github Actions : select the Emacs version
Running ERT tests suite with Cask and ert-runner	• To run all tests: `cask exec ert-runner` • To run a specific test: `cask exec ert-runner -p "name-of-specific-test"` • To run tests that are tagged fast or important: `cask exec ert-runner -t fast,important` • To run tests not tagged 'network': `cask exec ert-runner -t '! network'` • Combining options: `cask exec ert-runner test/moda-test.el -p "specific-functionality"`