

👤 master ▾


🌿 3 branches

🏷️ 0 tags

Go to file

Add file ▾

Code ▾

 alphapapa	Add: TODO	✓ d7d0768 12 days ago	🕒 306 commits
📁 data/57/f093b3-c2c9-436a-95d4-24...	Add: Elisp function walkthrough by Wilfred Hughes		2 months ago
📁 export	Change: Replace bigblow with darksun's minitoc-changes branch		3 years ago
📄 LICENSE	Initial commit		4 years ago
📄 README.org	Add: TODO		12 days ago
📄 dont-tread-on-emacs-150.png	Add flag		4 years ago
📄 epdh.el	Change: bench-multi macros		2 months ago
📄 index.html	Add: TODO		12 days ago

The Emacs Package Developer’s Handbook

After developing some little Emacs packages for a year or so, I began to notice how I’d forget little things that I learned, and then I’d have to go hunting for that information again. I also noticed how there are some issues for which there doesn’t seem to be a “best practice” or “Standard Operating Procedure” to refer to.



So this is intended to be a place to collect and organize information related to Emacs package development. Built with Emacs, by Emacs package developers, for Emacs package developers.

You can read this Org file directly on the [repository](#) rendered by GitHub (which lacks support for some features of the document, such as links between sections), or you can read the [HTML version](#).

Note: The primary sections are listed at the top of the page in the horizontal bar.

Contents

- [Emacs Lisp](#)
- [Blogs](#)
- [People](#)
- [Contributions](#)
- [Tasks](#)

Emacs Lisp

- [Animations / Screencasts](#)
- [Asynchronicity](#)
- [Auditing / Reviewing](#)
- [Binding](#)
- [Buffers](#)
- [Checkers / linters](#)
- [Collections \(lists, vectors, hash-tables, etc.\)](#)
- [Color](#)
- [Data structure](#)
- [Date / Time](#)
- [Debugging](#)
- [Destructuring](#)
- [Documentation](#)
- [Editing](#)
- [General](#)
- [Highlighting / font-locking](#)
- [Multiprocessing \(generators, threads\)](#)
- [Networking](#)
- [Packaging](#)
- [Pattern matching](#)
- [Processes \(incl. IPC, RPC\)](#)
- [Optimization](#)
- [Refactoring](#)
- [Regular expressions](#)
- [Strings](#)
- [Testing](#)

About

An Emacs package development handbook. Built with Emacs, by Emacs package developers, for Emacs package developers.

[#emacs](#) [#emacs-lisp](#)

📖 Readme

📄 GPL-3.0 License





Releases

No releases published

Packages

No packages published

Contributors 4

-  **alphapapa**
-  **DamienCassou** Damien Cassou
-  **sergeyklay** Sergei Iakovlev
-  **kijimaD** Kijima Daigo

Environments 1

🔗 github-pages Active

Languages

🟡 JavaScript 47.3%

🟣 CSS 26.8%

🟠 Emacs Lisp 25.9%

- [User interface](#)
- [Version control](#)
- [XML / HTML](#)

Note: Usable Emacs Lisp code snippets (not examples) are tangled to the file `epdh.el`, which may be found in the [repository](#). You could even install the file as a package with [quelpa-use-package](#), like this:

```
(use-package epdh
  :quelpa (epdh :fetcher github :repo "alphapapa/emacs-package-dev-handbook"))
```

Animations / Screencasts

- [Tools](#)
 - [Demonstration](#)
 - [Recording](#)

Tools

Demonstration

Tools to aid demonstrating use of Emacs.

keycast: [Show current command and its key in the mode line](#)

command-log-mode: [log commands to buffer](#)

Recording

Tools to record screencasts of Emacs.

emacs-gif-screencast

Most Emacs screencasts are done by first recording a video, then converting it to a GIF. It seems like a waste of resources considering the little animation Emacs features.

Most of the time, one user action equals one visual change. By exploiting this fact, this package tailors GIF-screencasting for Emacs and captures one frame per user action. It's much more efficient than capturing frames even at a frequency as low as 10 per second. You could roughly expect 3 to 10 times smaller files compared to videos of the same quality. (Benchmarks are welcome.)

Another neat perk of action-based captures is that it produces editable GIF files: programs such as Gifsicle or The GIMP can be used to edit the delays or rearrange frames. This is particularly useful to fix mistakes in “post-processing”, without having to record the demo all over again.

The author of this document can vouch for the fact that this package is the easiest, most powerful way to make screencast animations in Emacs!

Bashcaster

While `emacs-gif-screencast` should usually be your first choice, the way it works, recording one frame per Emacs command, isn't suitable for every case. For general use, your editor can recommend Bashcaster, which is an easy-to-use script that can record the whole screen or individual windows, to videos or GIFs.

Asynchronicity

See Multiprocessing (generators, threads).

Auditing / Reviewing

Auditing, reviewing, and analyzing source code.

Tools

comb: [Interactive grep annotation tool for manual static analysis](#)

Comb is a native Emacs Lisp solution to search, browse and annotate occurrences of regular expressions in files. The interactive interface allows to perform an exhaustive classification of all the results to rule out false positives and asses proper matches during manual static analysis.

Binding

- [Articles](#)
- [Libraries](#)
 - [dash.el](#)
 - [thunk](#)
- [Tools](#)
 - [Lexical binding](#)

Information related to variable scope and binding in elisp code (e.g. lexical vs. dynamic scope).

Articles

[Make Flet Great Again « null program](#)

:archive.today: <http://archive.today/T8dHM>
Chris Wellons explains how the old `cl` macro `flet` changes in its new `cl-lib` version, `cl-flet`, and how to use `cl-letf` to achieve the old functionality. It's a way to override functions in both lexical and dynamic scope, which is especially useful for unit testing.

Libraries

dash.el

thunk

Thunk provides functions and macros to delay the evaluation of forms.

Use `thunk-delay` to delay the evaluation of a form (requires lexical-binding), and `thunk-force` to evaluate it. The result of the evaluation is cached, and only happens once.

Here is an example of a form which evaluation is delayed:

```
(setq delayed (thunk-delay (message "this message is delayed")))
```

`delayed` is not evaluated until `thunk-force` is called, like the following:

```
(thunk-force delayed)
```

This file also defines macros `thunk-let` and `thunk-let*` that are analogous to `let` and `let*` but provide lazy evaluation of bindings by using thunks implicitly (i.e. in the expansion).

Tools

Lexical binding

Buffers

- Articles
 - Buffer-Passing Style
- Best practices
 - Accessing buffer-local variables
 - Inserting strings
- Libraries

Articles

Buffer-Passing Style

Best practices

Accessing buffer-local variables

It's much faster to use `buffer-local-value` than `with-current-buffer` to access the value of a variable in a buffer.

```
(bench-multi :times 1000 :ensure-equal t
  :forms (("buffer-local-value" (--filter (equal 'magit-status-mode (buffer-local-value 'major-mode
                                              (buffer-list)))
      ("with-current-buffer" (--filter (equal 'magit-status-mode (with-current-buffer it
                                                                  major-mode))
                                      (buffer-list)))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
buffer-local-value	50.34	0.047657734	0	0.0
with-current-buffer	slowest	2.399323452	0	0.0

Inserting strings

Inserting strings into buffers with `insert` is generally fast, but it can slow down in buffers with lots of markers or overlays. In general, it can be faster to insert one large string (which may include newlines). For example:

```
(let ((strings (cl-loop for i from 1 to 1000
                        collect (number-to-string i))))
  (garbage-collect)
  (--sort (< (caddr it) (caddr other))
    (cl-loop for times in '(1 10 100)
      append (a-list "(loop do (insert ..."
                    (cons times
                        (benchmark-run-compiled times
                          (with-temp-buffer
                            (cl-loop for string in strings
                                do (insert string))))))
                    "(apply #'insert ..."
                    (cons times
                        (benchmark-run-compiled times
                          (with-temp-buffer
                            (apply #'insert strings))))
                    "(insert (apply #'concat ..."
                    (cons times
                        (benchmark-run-compiled times
```

```
(with-temp-buffer
  (insert (apply #'concat strings)))))))))
```

#+RESULTS[aa866ca87dbf71476c735ed51fca7373934bbf4f]:

(insert (apply #'concat ...	100	0.000142085	0	0.0
(insert (apply #'concat ...	10	0.000161172	0	0.0
(insert (apply #'concat ...	1	0.00018764	0	0.0
(apply #'insert ...	10	0.000665472	0	0.0
(apply #'insert ...	100	0.000678471	0	0.0
(apply #'insert ...	1	0.000755329	0	0.0
(loop do (insert ...	10	0.000817031	0	0.0
(loop do (insert ...	100	0.000869779	0	0.0
(loop do (insert ...	1	0.001490397	0	0.0

The fastest method here is to call `insert` once with the result of calling `concat` once, using `apply` to pass all of the strings. With 100 iterations, it's about 6x faster than the next-fastest method, and even with 1 iteration, it's over 2x faster.

Libraries

[m-buffer-el: List Oriented Buffer Operations](#)

[bui.el: Buffer interface library](#)

BUI (Buffer User Interface) is an Emacs library that can be used to make user interfaces to display some kind of entries (like packages, buffers, functions, etc.). The intention of BUI is to be a high-level library which is convenient to be used both by:

package makers, as there is no need to bother about implementing routine details and usual features (like buffer history, filtering displayed entries, etc.);

users, as it provides familiar and intuitive interfaces with usual keys (for moving by lines, marking, sorting, switching between buttons); and what is also important, the defined interfaces are highly configurable through various generated variables.

Checkers / linters

[elisp-lint: basic linting for Emacs Lisp](#)

Includes the following validators:

- [byte-compile](#)
- [check-declare](#)
- [checkdoc](#)
- [fill-column](#)
- [indent](#)
- [indent-character](#)
- [package-format](#)
- [trailing-whitespace](#)

[flycheck-package: Flycheck checker for Emacs package metadata](#)

This library provides a `flycheck` checker for the metadata in Emacs Lisp files which are intended to be packages. That metadata includes the package description, its dependencies and more. The checks are performed by the separate `package-lint` library.

[relint: regexp lint tool](#)

Relint scans elisp files for mistakes in regexps, including deprecated syntax and bad practice. It also checks the regexp-like arguments to `skip-chars-forward`, `skip-chars-backward`, `skip-syntax-forward` and `skip-syntax-backward`.

Collections (lists, vectors, hash-tables, etc.)

- [Best practices](#)
- [Examples](#)
- [Libraries](#)
- [Tools](#)

Best practices

- [Collecting items into a list](#)
- [Diffing two lists](#)
- [Filtering a list](#)
- [Looking up associations](#)

Collecting items into a list

Here are some examples of fast ways to collect items into a list.

```
(bench-multi-lexical :times 500000 :ensure-equal t
:forms (("cl-loop" (let ((l '(1 2 3 4)))
  (cl-loop for val in l
    collect val)))
  ("push-nreverse with setf/pop" (let ((l '(1 2 3 4))
    val r)
    (while (setf val (pop l))
      (push val r))
    (nreverse r)))
  ("push-nreverse with when-let*/pop" (let ((l '(1 2 3 4))
    r)
    (while (when-let* ((val (pop l)))
      (push val r)))
    (nreverse r)))
  ("nconc with when-let*/pop" (let ((l '(1 2 3 4))
    r)
    (while (when-let* ((val (pop l)))
      (setf r (nconc r (list val)))))
    r))
  ("nconc with setf/pop" (let ((l '(1 2 3 4))
    val r)
    (while (setf val (pop l))
      (setf r (nconc r (list val))))
    r))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
cl-loop	1.01	0.154578	0	0
push-nreverse with setf/pop	1.02	0.155930	0	0
push-nreverse with when-let*/pop	1.23	0.159211	0	0
nconc with setf/pop	1.06	0.195685	0	0
nconc with when-let*/pop	slowest	0.207103	0	0

As is usually the case, the `cl-loop` macro expands to the most efficient code, which uses `(setf val (car ... , push , and nreverse :`

```
(cl-block nil
  (let* ((--cl-var-- l)
    (val nil)
    (--cl-var-- nil))
    (while (consp --cl-var--)
      (setf val (car --cl-var--))
      (push val --cl-var--)
      (setf --cl-var-- (cdr --cl-var--)))
    (nreverse --cl-var--)))
```

However, in some cases `cl-loop` may expand to code which uses `nconc` , which, as the benchmark shows, is much slower. In that case, you may write the loop without `cl-loop` to avoid using `nconc` .

Diffing two lists

As expected, `seq-difference` is the slowest, because it's a generic function that dispatches based on the types of its arguments, which is relatively slow in Emacs. And it's not surprising that `cl-nset-difference` is generally slightly faster than `cl-set-difference` , since it's destructive.

However, it is surprising how much faster `-difference` is than `cl-nset-difference` .

It's also nonintuitive that `-difference` suffers a large performance penalty by binding `-compare-fn` (the equivalent of the `:test` argument to `cl-set-difference`): while one might expect that setting it to `string=` would give a slight performance increase, it's actually faster to let `-difference` use its default, `equal` .

Note that since this benchmark compares lists of strings, `cl-nset-difference` requires setting the `:test` argument, since it uses `eq` by default, which does not work for comparing strings.

```
(defmacro test/set-lists ()
  `(setf list1 (cl-loop for i from 0 below 1000
    collect (number-to-string i))
    list2 (cl-loop for i from 500 below 1500
      collect (number-to-string i))))

(let (list1 list2)
  (bench-multi-lexical :times 10 :ensure-equal t
:forms ((" -difference"
  (progn
    (test/set-lists)
    (-difference list1 list2)))
  (" -difference string="
  (progn
    ;; This is much slower because of the way '-contains?'
    ;; works when '-compare-fn' is non-nil.
    (test/set-lists)
    (let ((-compare-fn #'string=))
      (-difference list1 list2))))
  ("cl-set-difference equal"
  (progn
    (test/set-lists)
    (cl-set-difference list1 list2 :test #'equal)))))
```

```

("cl-set-difference string="
 (progn
  (test/set-lists)
  (cl-set-difference list1 list2 :test #'string=)))
("cl-nset-difference equal"
 (progn
  (test/set-lists)
  (cl-nset-difference list1 list2 :test #'equal)))
("cl-nset-difference string="
 (progn
  (test/set-lists)
  (cl-nset-difference list1 list2 :test #'string=)))
("seq-difference"
 (progn
  (test/set-lists)
  (seq-difference list1 list2)))
("seq-difference string="
 (progn
  (test/set-lists)
  (seq-difference list1 list2 #'string=))))))

```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
-difference	7.16	0.084484	0	0
cl-nset-difference equal	1.05	0.605193	0	0
cl-set-difference string=	1.01	0.636973	0	0
cl-set-difference equal	1.01	0.644919	0	0
cl-nset-difference string=	1.19	0.650708	0	0
-difference string=	1.59	0.773919	0	0
seq-difference	1.05	1.232616	0	0
seq-difference string=	slowest	1.293030	0	0

Filtering a list

Using `-select` from `dash.el` seems to be the fastest way:

```

(let ((list (cl-loop for i from 1 to 1000
                    collect i)))
  (bench-multi :times 100
   :ensure-equal t
   :forms ((("-non-nil (--map (when ..." (-non-nil
                                           (--map (when (cl-evenp it) it) list)))
            ("(delq nil (--map (when ..." (delq nil
                                           (--map (when (cl-evenp it) it) list)))
            ("cl-loop" (cl-loop for i in list
                               when (cl-evenp i)
                               collect i))
            ("-select" (-select #'cl-evenp list))
            ("cl-remove-if-not" (cl-remove-if-not #'cl-evenp list))
            ("seq-filter" (seq-filter #'cl-evenp list)))))))

```

#+RESULTS[6b2e97c1ehead84a53fd771684cc3e155e7f6b1e]:

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
-select	1.17	0.01540391	0	0.0
cl-loop	1.05	0.01808226	0	0.0
seq-filter	1.13	0.01891708	0	0.0
(delq nil (--map (when ...	1.15	0.02134727	0	0.0
cl-remove-if-not	1.18	0.02459478	0	0.0
(-non-nil (--map (when ...	slowest	0.02903999	0	0.0

Looking up associations

There are a few options in Emacs Lisp for looking up values in associative data structures: association lists (alists), property lists (plist), and hash tables. Which one performs best in a situation may depend on several factors. This benchmark shows what may be a common case: looking up values using a string as the key. We compare several combinations, including the case of prepending a prefix to the string, internng it, and looking up the resulting symbol (which might be done, e.g. when looking up a function to call based on the value of a string).

```

(bench-multi-lets :times 10000 :ensure-equal t
 :lets (("with 26 pairs"
        ((char-range (cons ?A ?Z))
         (strings (cl-loop for char from (car char-range) to (cdr char-range)
                          collect (concat "prefix-" (char-to-string char))))
         (strings-alist (cl-loop for string in strings
                                collect (cons string string)))
         (symbols-alist (cl-loop for string in strings
                                collect (cons (intern string) string)))
         (strings-plist (map-into strings-alist 'plist))
         (symbols-plist (map-into symbols-alist 'plist))
         (strings-ht (map-into strings-alist '(hash-table :test equal)))
         (symbols-ht-equal (map-into symbols-alist '(hash-table :test equal)))
         (symbols-ht-eq (map-into symbols-alist '(hash-table :test eq))))
        ("with 52 pairs"

```

```

(char-range (cons ?A ?z))
(strings (cl-loop for char from (car char-range) to (cdr char-range)
  collect (concat "prefix-" (char-to-string char))))
(strings-alist (cl-loop for string in strings
  collect (cons string string)))
(symbols-alist (cl-loop for string in strings
  collect (cons (intern string) string)))
(strings-plist (map-into strings-alist 'plist))
(symbols-plist (map-into symbols-alist 'plist))
(strings-ht (map-into strings-alist '(hash-table :test equal)))
(symbols-ht-equal (map-into symbols-alist '(hash-table :test equal)))
(symbols-ht-eq (map-into symbols-alist '(hash-table :test eq))))
:forms (("strings/alist-get/string=" (sort (cl-loop for string in strings
  collect (alist-get string strings-alist nil nil
    #'string<))
  ("strings/plist" (sort (cl-loop for string in strings
    collect (plist-get strings-plist string))
    #'string<))
  ("symbols/concat/intern/plist" (sort (cl-loop for char from (car char-range) to (cdr char-
    for string = (concat "prefix-" (char-to-stri
    for symbol = (intern string)
    collect (plist-get symbols-plist symbol))
    #'string<))
  ("strings/alist-get/equal" (sort (cl-loop for string in strings
    collect (alist-get string strings-alist nil nil
    #'string<))
  ("strings/hash-table/equal" (sort (cl-loop for string in strings
    collect (gethash string strings-ht))
    #'string<))
  ("symbols/concat/intern/hash-table/equal" (sort (cl-loop for char from (car char-range) to
    for string = (concat "prefix-" (c
    for symbol = (intern string)
    collect (gethash symbol symbols-h
    #'string<))
  ("symbols/concat/intern/hash-table/eq" (sort (cl-loop for char from (car char-range) to (c
    for string = (concat "prefix-" (char
    for symbol = (intern string)
    collect (gethash symbol symbols-ht-e
    #'string<))
  ("symbols/concat/intern/alist-get" (sort (cl-loop for char from (car char-range) to (cdr c
    for string = (concat "prefix-" (char-to-
    for symbol = (intern string)
    collect (alist-get symbol symbols-alist)
    #'string<))
  ("symbols/concat/intern/alist-get/equal" (sort (cl-loop for char from (car char-range) to
    for string = (concat "prefix-" (ch
    for symbol = (intern string)
    collect (alist-get symbol symbols-
    #'string<))))))

```

#+RESULTS[041dd7c6644612027379e3558fc60e61eb4896a]:

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
with 26 pairs: strings/hash-table/equal	1.06	0.040321	0	0
with 26 pairs: strings/plist	2.26	0.042848	0	0
with 52 pairs: strings/hash-table/equal	1.27	0.096877	0	0
with 26 pairs: strings/alist-get/equal	1.04	0.123039	0	0
with 26 pairs: strings/alist-get/string=	1.03	0.128221	0	0
with 52 pairs: strings/plist	2.62	0.131451	0	0
with 26 pairs: symbols/concat/intern/hash-table/eq	1.00	0.344524	1	0.266744
with 26 pairs: symbols/concat/intern/hash-table/equal	1.01	0.344951	1	0.267860
with 26 pairs: symbols/concat/intern/plist	1.02	0.349360	1	0.266529
with 26 pairs: symbols/concat/intern/alist-get	1.19	0.358071	1	0.267457
with 26 pairs: symbols/concat/intern/alist-get/equal	1.11	0.424895	1	0.271568
with 52 pairs: strings/alist-get/equal	1.03	0.471979	0	0
with 52 pairs: strings/alist-get/string=	1.50	0.485663	0	0
with 52 pairs: symbols/concat/intern/hash-table/equal	1.00	0.730628	2	0.547082
with 52 pairs: symbols/concat/intern/hash-table/eq	1.05	0.733726	2	0.548910
with 52 pairs: symbols/concat/intern/alist-get	1.00	0.773320	2	0.545707
with 52 pairs: symbols/concat/intern/plist	1.36	0.774225	2	0.549963
with 52 pairs: symbols/concat/intern/alist-get/equal	slowest	1.056641	2	0.545522

We see that hash-tables are generally the fastest solution.

Comparing alists and plists, we see that, when using string keys, plists are significantly faster than alists, even with 52 pairs. When using symbol keys, plists are faster with 26 pairs; with 52, plists and alists (using `alist-get` with `eq` as the test function) are nearly the same in performance.

Also, perhaps surprisingly, when looking up a string in an alist, using `equal` as the test function may be faster than using the type-specific `string=` function (possibly indicating an optimization to be made in Emacs's C code).

Compare looking up interned symbols in obarray instead of hash table

Compare a larger number of pairs

Examples

Alists

Creation

```
;;; Built-in methods

(list (cons 'one 1) (cons 'two 2)) ;; => ((one . 1) (two . 2))

'((one . 1) (two . 2)) ;; => ((one . 1) (two . 2))

(let ((numbers (list)))
  (map-put numbers 'one 1)
  (map-put numbers 'two 2)) ;; => ((two . 2) (one . 1))

;;; Packages

;; 'a-list' from a.el is the best way to make a new alist.

(a-list 'one 1
        'two 2) ;; => ((one . 1) (two . 2))
```

Adding to

Single elements

```
;;; Built-in methods

;; 'map-put' is the best built-in way. Requires Emacs 25.1+.

(let ((numbers (list (cons 'one 1))))
  (map-put numbers 'two 2)
  numbers) ;; => ((two . 2) (one . 1))

;; More primitive methods

;; Not recommended, but not too complicated:
(let ((numbers (list (cons 'one 1)))
      (more-numbers (a-list 'two 2
                            'three 3)))
  (append numbers more-numbers)) ;; => ((one . 1) (two . 2) (three . 3))

;; Don't do it this way, but it does demonstrate list/cons-cell
;; structure:
(let ((numbers (list (cons 'one 1)))
      (cons (cons 'three 3)
             (cons (cons 'two 2)
                   numbers)))
  numbers)) ;; => ((three . 3) (two . 2) (one . 1))
```

Multiple elements

```
;;; Built-in methods

;; 'map-merge': if you're restricted to built-in packages, this works
;; well (requires Emacs 25.1+):
(let ((numbers (list (cons 'one 1)))
      (more-numbers (a-list 'two 2
                            'three 3)))
  (map-merge 'list numbers more-numbers)) ;; => ((three . 3) (two . 2) (one . 1))

;; Without map.el, you could use 'append':
(let ((numbers (list (cons 'one 1)))
      (more-numbers (a-list 'two 2
                            'three 3)))
  (append numbers more-numbers)) ;; => ((one . 1) (two . 2) (three . 3))

;;; Packages

;; 'a-merge' from a.el is probably the best way:
(let ((numbers (list (cons 'one 1)))
      (more-numbers (a-list 'two 2
                            'three 3)))
  (a-merge numbers more-numbers)) ;; => ((three . 3) (two . 2) (one . 1))
```

Property lists (plists)

Removing properties

[According to Stefan Monnier:](#)

The plist design in Emacs was based on the idea that you shouldn't distinguish an entry with a nil value from an entry that's absent. Hence `plist-remove` is not needed because you can do `(plist-put PLIST PROP nil)` instead.

Of course, in the mean time, `plist-member` appeared, breaking the design, so this answer is probably not 100% satisfactory, but I still think you'll generally be better off if you can ignore the difference between `nil` and "absent".

If you do need to remove a key and value from a plist, you could use `cl-remf` or `map-delete` (the former probably being faster).

Libraries

a.el: functions for dealing with association lists and hash tables. Inspired by Clojure.

asoc.el: alist library

emacs-kv: key/value collection-type functions, for alists, hash tables and plists

ht.el: The missing hash table library

This library provides a consistent and comprehensive set of functions for working with hash tables: they're named consistently, take a natural and consistent argument order, and cover operations that the standard Emacs functions don't.

list-utils: List-manipulation utility functions

Similar to `dash.el`, but with slightly different behavior that may be useful, and some unique features. These functions are provided:

<code>make-tconc</code>	<code>list-utils-depth</code>
<code>tconc-p</code>	<code>list-utils-flat-length</code>
<code>tconc-list</code>	<code>list-utils-flatten</code>
<code>tconc</code>	<code>list-utils-alist-or-flat-length</code>
<code>list-utils-cons-cell-p</code>	<code>list-utils-alist-flatten</code>
<code>list-utils-cyclic-length</code>	<code>list-utils-insert-before</code>
<code>list-utils-improper-p</code>	<code>list-utils-insert-after</code>
<code>list-utils-make-proper-copy</code>	<code>list-utils-insert-before-pos</code>
<code>list-utils-make-proper-inplace</code>	<code>list-utils-insert-after-pos</code>
<code>list-utils-make-improper-copy</code>	<code>list-utils-and</code>
<code>list-utils-make-improper-inplace</code>	<code>list-utils-not</code>
<code>list-utils-linear-p</code>	<code>list-utils-xor</code>
<code>list-utils-linear-subseq</code>	<code>list-utils-uniq</code>
<code>list-utils-cyclic-p</code>	<code>list-utils-dupes</code>
<code>list-utils-cyclic-subseq</code>	<code>list-utils-singlets</code>
<code>list-utils-make-linear-copy</code>	<code>list-utils-partition-dupes</code>
<code>list-utils-make-linear-inplace</code>	<code>list-utils-plist-reverse</code>
<code>list-utils-safe-length</code>	<code>list-utils-plist-del</code>
<code>list-utils-safe-equal</code>	

map.el: Map manipulation functions

`map` is included with Emacs, but the latest version, which may include improvements since the last Emacs release, is now available separately on [GNU ELPA](#).

stream: Lazy sequences

`stream.el` provides an implementation of streams, implemented as delayed evaluation of cons cells.

Functions defined in `seq.el` can also take a stream as input.

Streams could be created from any sequential input data:

- sequences, making operation on them lazy
- a set of 2 forms (first and rest), making it easy to represent infinite sequences
- buffers (by character)
- buffers (by line)
- buffers (by page)
- IO streams
- orgmode table cells
- ...

persist.el: Persist variables between sessions

This package provides variables which persist across sessions.

The main entry point is ``persist-defvar`` which behaves like ``defvar`` but which persists the variables between session. Variables are automatically saved when Emacs exits.

Other useful functions are ``persist-save`` which saves the variable immediately, ``persist-load`` which loads the saved value, ``persist-reset`` which resets to the default value.

Values are stored in a directory in ``user-emacs-directory``, using one file per value. This makes it easy to delete or remove unused variables.

Tools

let-alist

=with-dict=, =with-plist-vals=

Color

Libraries

yk-color: Linear RGB color manipulation

Includes these functions:

- `yk-color-adjust`
- `yk-color-adjust-rgb`
- `yk-color-blend`
- `yk-color-blend-rgb`
- `yk-color-contrast-ratio`
- `yk-color-contrast-ratio-rgb`
- `yk-color-from-rgb`
- `yk-color-relative-luminance`
- `yk-color-relative-luminance-rgb`
- `yk-color-rgb-to-srgb`
- `yk-color-srgb-to-rgb`
- `yk-color-to-rgb`

Data structure

- [Articles](#)
- [Libraries](#)

Articles

Options for Structured Data in Emacs Lisp « null program

:archive.today: <http://archive.today/YxwP5>

Toby 'qubit' Cubitt: Data structure packages

Individual libraries from this article are listed below.

Libraries

- [heap.el](#)
- [myers](#)
- [queue.el](#)
- [Tagged Non-deterministic Finite state Automata \(tNFA.el\)](#)
- [trie.el](#)
- [dict-tree.el](#)
- [extmap](#): Externally-stored constant mapping

[heap.el](#)

A heap is a form of efficient self-sorting tree. In particular, the root node is guaranteed to be the highest-ranked entry in the tree. (The comparison function used for ranking the data can, of course, be freely defined). They are often used as priority queues, for scheduling tasks in order of importance, and for implementing efficient sorting algorithms (such as heap-sort).

[myers](#)

This package implements Eugene W. Myers's "stacks" which are like standard singly-linked lists, except that they also provide efficient lookup. More specifically:

`cons/car/cdr` are $O(1)$, while `(nthcdr N L)` is $O(\min(N, \log L))$

For details, see "[An applicative random-access stack](#)", Eugene W. Myers, 1983, [Information Processing Letters](#).

[queue.el](#)

A queue can be used both as a first-in last-out and as a first-in first-out stack, i.e. elements can be added to and removed from the front or back of the queue. (This library is an updated re-implementation of the old Elib queue library.)

Tagged Non-deterministic Finite state Automata (tNFA.el)

Features of modern regexp implementations, including Emacs', mean they can recognise much more than regular languages. This comes with a big downside: matching certain pathological regexps is very time-consuming. (In fact, it's NP-complete.)

A tagged, non-deterministic finite state automata (NFA) is an abstract computing machine that recognises regular languages. In layman's terms, they are used to decide whether a string matches a regular expression. The "tagged" part lets the NFA do group-capture: it returns information about which parts of a string matched which subgroup of the regular expression.

Why re-implement regular expression matching when Emacs comes with extensive built-in support for regexps? Primarily, because some algorithms require access to the NFA states produced part way through the regular expression matching process. Secondly, because Emacs regexps only work on strings, whereas regular expressions can equally well be used to match other Lisp sequence types.

trie.el

A trie stores data associated with "strings" (not necessarily the string data type; any ordered sequence of elements can be used). It stores them in such a way that both storage size and data lookup are reasonably space- and time- efficient, respectively. But, more importantly, advanced string queries are also very efficient, such as finding all strings with a given prefix, finding approximate matches, finding all strings matching a regular expression, returning results in alphabetical or any other order, returning only the first few results, etc.

dict-tree.el

The dictionary tree data structures are a hybrid between tries and hash tables. Data is stored in a trie, but results that take particularly long to retrieve are cached in hash tables, which are automatically synchronised with the trie. The dictionary package provides persistent storage of the data structures in files, and many other convenience features.

extmap: Externally-stored constant mapping

extmap is a very simple package that lets you build a read-only, constant database that maps Elisp symbols to almost arbitrary Elisp objects. The idea is to avoid preloading all data to memory and only retrieve it when needed. This package doesn't use any external programs, making it a suitable dependency for smaller libraries.

Date / Time

Libraries

ts.el: Timestamp and date-time library

ts aids in parsing, formatting, and manipulating timestamps.

ts is a date and time library for Emacs. It aims to be more convenient than patterns like `(string-to-number (format-time-string "%Y"))` by providing easy accessors, like `(ts-year (ts-now))`.

To improve performance (significantly), formatted date parts are computed lazily rather than when a timestamp object is instantiated, and the computed parts are then cached for later access without recomputing. Behind the scenes, this avoids unnecessary `(string-to-number (format-time-string... calls, which are surprisingly expensive.`

Examples

Get parts of the current date:

```
;; When the current date is 2018-12-08 23:09:14 -0600:
(ts-year (ts-now))      ;=> 2018
(ts-month (ts-now))     ;=> 12
(ts-day (ts-now))       ;=> 8
(ts-hour (ts-now))      ;=> 23
(ts-minute (ts-now))    ;=> 9
(ts-second (ts-now))    ;=> 14
(ts-tz-offset (ts-now)) ;=> "-0600"

(ts-dow (ts-now))       ;=> 6
(ts-day-abbr (ts-now))  ;=> "Sat"
(ts-day-name (ts-now))  ;=> "Saturday"

(ts-month-abbr (ts-now)) ;=> "Dec"
(ts-month-name (ts-now)) ;=> "December"

(ts-tz-abbr (ts-now))   ;=> "CST"
```

Increment the current date:

```
;; By 10 years:
(list :now (ts-format)
      :future (ts-format (ts-adjust 'year 10 (ts-now))))
;=> ( :now "2018-12-15 22:00:34 -0600"
     :future "2028-12-15 22:00:34 -0600")

;; By 10 years, 2 months, 3 days, 5 hours, and 4 seconds:
(list :now (ts-format)
      :future (ts-format
                 (ts-adjust 'year 10 'month 2 'day 3
                             'hour 5 'second 4
                             (ts-now))))
```

```
;;=> ( :now "2018-12-15 22:02:31 -0600"
;;      :future "2029-02-19 03:02:35 -0600")
```

What day of the week was 2 days ago?

```
(ts-day-name (ts-dec 'day 2 (ts-now))) ;=> "Thursday"

;; Or, with threading macros:
(thread-last (ts-now) (ts-dec 'day 2) ts-day-name) ;=> "Thursday"
(->> (ts-now) (ts-dec 'day 2) ts-day-name) ;=> "Thursday"
```

Get timestamp for this time last week:

```
(ts-unix (ts-adjust 'day -7 (ts-now)))
;;=> 1543728398.0

;; To confirm that the difference really is 7 days:
(/ (- (ts-unix (ts-now))
      (ts-unix (ts-adjust 'day -7 (ts-now)))))
86400)
;;=> 7.000000567521762

;; Or human-friendly as a list:
(ts-human-duration
 (ts-difference (ts-now)
                (ts-dec 'day 7 (ts-now))))
;;=> (:years 0 :days 7 :hours 0 :minutes 0 :seconds 0)

;; Or as a string:
(ts-human-format-duration
 (ts-difference (ts-now)
                (ts-dec 'day 7 (ts-now))))
;;=> "7 days"

;; Or confirm by formatting:
(list :now (ts-format)
      :last-week (ts-format (ts-dec 'day 7 (ts-now))))
;;=> ( :now "2018-12-08 23:31:37 -0600"
;;      :last-week "2018-12-01 23:31:37 -0600")
```

emacs-datetime

The primary function provided is: (datetime-format SYM-OR-FMT &optional TIME &rest OPTION)

```
(datetime-format "%Y-%m-%d") ;=> "2018-08-22"
(datetime-format 'atom) ;=> "2018-08-22T18:23:47-05:00"
(datetime-format 'atom "2112-09-03 00:00:00" :timezone "UTC") ;=> "2112-09-03T00:00:00+00:00"
```

There are several other symbols provided besides `atom`, such as `rfc-3339`, which formats dates according to that RFC.

Debugging

- [Tools](#)
 - [Edebug](#)
 - [debug-warn macro](#)

Tools

Edebug

Edebug is a built-in stepping debugger in Emacs. It's [thoroughly documented in the `elisp` manual](#).

Declaring debug forms for keyword arguments

Declaring `debug` forms for functions and macros that take keyword arguments can be confusing. Here's a contrived example:

```
(cl-defmacro make-fn (name docstring &key args bindings body)
  (declare (indent defun)
    (debug (&define symbolp stringp
                  &rest [&or ["body" def-form] [keywordp listp]])))
  `(defun ,name ,args
    ,docstring
    (let* ,bindings
      ,body)))

(make-fn my-fn
  "This is my function."
  :bindings ((one 1)
             (two 2))
  :body (list one two))
```

Submit this as an improvement to the [Elisp manual](#)

Probably should first replace the `:bindings` part with this, which correctly matches `let bindings: (&rest &or symbolp (gate symbolp &optional def-form))`.

debug-warn macro

This macro simplifies print-style debugging by automatically including the names of the containing function and argument forms, rather than requiring the programmer to write `format` strings manually. If `warning-minimum-log-level` is not `:debug` at expansion time, the macro expands to nil, which the byte-compiler eliminates, so in byte-compiled code, the macro has no overhead at runtime when not debugging. In interpreted code, the overhead when not debugging is minimal. When debugging, the expanded form also returns nil so, e.g. it may be used in a conditional in place of nil. The macro is tangled to `epdh.el`, but it may also be added directly to source files.

For example, when used like:

```
(eval-and-compile
 (setq-local warning-minimum-log-level :debug))

(defun argh (var)
  (debug-warn (current-buffer) "This is bad!" (point) var)
  var)

(Argb 1)
```

This warning would be shown in the `*warnings*` buffer:

```
Debug (Argb): (CURRENT-BUFFER):*scratch* This is bad! (POINT):491845 VAR:1
```

But if `warning-minimum-log-level` had any other value, and the buffer were recompiled, there would be no output, and none of the arguments to the macro would be evaluated at runtime.

It may even be used in place of comments! For example, instead of:

```
(if (foo-p thing)
    ;; It's a foo: frob it.
    (frob thing)
    ;; Not a foo: flub it.
    (flub thing))
```

You could write:

```
(if (foo-p thing)
    (progn
      (debug-warn "It's a foo: frob it." thing)
      (frob thing))
    (debug-warn "Not a foo: flub it." thing)
    (flub thing))
```

```
;; To make newer versions of `map' load for the `pcase' pattern.
(require 'map)
```

```
(cl-defmacro epdh/debug-warn (&rest args)
  "Display a debug warning showing the runtime value of ARGS.
The warning automatically includes the name of the containing
function, and it is only displayed if `warning-minimum-log-level'
is `:debug' at expansion time (otherwise the macro expands to nil
and is eliminated by the byte-compiler). When debugging, the
form also returns nil so, e.g. it may be used in a conditional in
place of nil."
```

Each of `ARGS` may be a string, which is displayed as-is, or a symbol, the value of which is displayed prefixed by its name, or a Lisp form, which is displayed prefixed by its first symbol.

Before the actual `ARGS` arguments, you can write keyword arguments, i.e. alternating keywords and values. The following keywords are supported:

```
:buffer BUFFER    Name of buffer to pass to `display-warning'.
:level LEVEL      Level passed to `display-warning', which see.
                  Default is :debug."
;; TODO: Can we use a compiler macro to handle this more elegantly?
(pcase-let* ((fn-name (when byte-compile-current-buffer
                        (with-current-buffer byte-compile-current-buffer
                          ;; This is a hack, but a nifty one.
                          (save-excursion
                            (beginning-of-defun)
                            (cl-second (read (current-buffer)))))))
  (plist-args (cl-loop while (keywordp (car args))
                      collect (pop args)
                      collect (pop args)))
  ((map (:buffer buffer) (:level level)) plist-args)
  (level (or level :debug))
  (string (cl-loop for arg in args
                  concat (pcase arg
                          ((pred stringp) "%S ")
                          ((pred symbolp)
                           (concat (upcase (symbol-name arg)) "%S "))
                          ((pred listp)
                           (concat "(" (upcase (symbol-name (car arg)))
                                   (pcase (length arg)
                                     (1 " ")
                                     (_ "..."))
                                   "%S "))))))

  (when (eq :debug warning-minimum-log-level)
    `(let ((fn-name ,if fn-name
                      `',fn-name
                      ;; In an interpreted function: use `backtrace-frame' to get the
                      ;; function name (we have to use a little hackery to figure out
                      ;; how far up the frame to look, but this seems to work).
```

```

      `(cl-loop for frame in (backtrace-frames)
        for fn = (cl-second frame)
        when (not (or (subrp fn)
                      (special-form-p fn)
                      (eq 'backtrace-frames fn)))
        return (make-symbol (format "%s [interpreted]" fn))))))
      (display-warning fn-name (format ,string ,@args) ,level ,buffer)
      nil)))

```

Destructuring

See Pattern matching.

Documentation

Tools

ox-texinfo+

This is helpful when exporting Org files to Info manuals.

This package provides some extensions for Org's `texinfo` exporter defined in `ox-texinfo`.

1. Create `@deffn` and similar definition items by writing list items in Org that look similar to what they will look like in Info.
2. Optionally share a section's node with some or all of its child sections.
3. Optionally modify the Org file before exporting it.
4. Fully respect the local value of `indent-tabs-mode` from the Org file when editing source blocks and exporting. This affects all source blocks and all exporters.

Editing

- [Tools](#)
 - [aggressive-indent-mode](#): minor mode that keeps your code always indented
 - [beginend.el](#)
 - [expand-region.el](#): Increase selected region by semantic units
 - [helm-navi](#): Navigate file sections and language keywords using Helm
 - [iedit](#): Edit multiple regions simultaneously in a buffer or a region
 - [lispy](#): short and sweet LISP editing
 - [multi-line](#): multi-line everything from function invocations and definitions to array and map literals in a wide variety of languages
 - [multiple-cursors.el](#): Multiple cursors
 - [smartparens](#): Minor mode that deals with parens pairs and tries to be smart about it

Tools

[aggressive-indent-mode](#): minor mode that keeps your code always indented

[beginend.el](#)

This package, by Damien Cassou and Matus Goljer, helps navigation by redefining the `M-<` and `M->` keys do, depending on the major-mode.

[expand-region.el](#): Increase selected region by semantic units

[helm-navi](#): Navigate file sections and language keywords using Helm

[iedit](#): Edit multiple regions simultaneously in a buffer or a region

`iedit` makes it easy to rename symbols within a function or in a whole buffer. Simply activate `iedit-mode` with point on a symbol, and it will be highlighted in the chosen scope, and any changes you make to the symbol are made in each highlighted occurrence. It's like a smart, purposeful version of `multiple-cursors`.

The editor of this handbook uses `iedit` with these customizations:

[ap/iedit-or-flyspell](#)

Globally bound to `C-;`. In a `prog-mode`-derived buffer, either corrects the last misspelled word with `flyspell` when point is in a comment or string, or activates `iedit-mode`. In non-`prog-mode`-derived buffers, corrects with `flyspell`.

```

(defun ap/iedit-or-flyspell ())
  "Call 'iedit-mode' or correct misspelling with flyspell, depending..."
  (interactive)
  (if (or iedit-mode
          (and (derived-mode-p 'prog-mode)
                (not (or (nth 4 (syntax-ppss))
                        (nth 3 (syntax-ppss)))))
          ;; prog-mode is active and point is in a comment, string, or
          ;; already in iedit-mode
          (call-interactively #'ap/iedit-mode)
          ;; Not prog-mode or not in comment or string
          (if (not (equal flyspell-previous-command this-command))
              ;; FIXME: This mostly works, but if there are two words on the
              ;; same line that are misspelled, it doesn't work quite right
              ;; when correcting the earlier word after correcting the later

```

```
;; one

;; First correction; autocorrect
(call-interactively 'flyspell-auto-correct-previous-word)
;; First correction was not wanted; use popup to choose
(progn
  (save-excursion
    (undo)) ; This doesn't move point, which I think may be the problem.
    (flyspell-region (line-beginning-position) (line-end-position))
    (call-interactively 'flyspell-correct-previous-word-generic))))
```

ap/iedit-mode

Calls `iedit-mode` with function-local scope by default, or global scope when called with a universal prefix.

```
(defun ap/iedit-mode (orig-fn)
  "Call `iedit-mode' with function-local scope by default, or global scope if called with a universa
  (interactive)
  (pcase current-prefix-arg
    ('nil (funcall orig-fn '(0)))
    ('(4) (funcall orig-fn))
    (_ (user-error "`ap/iedit-mode' called with prefix: %s" prefix))))

;; Override default `iedit-mode' function with advice.
(advice-add #'iedit-mode :around #'ap/iedit-mode)
```

Helpful minibuffer message

Confirms when an `iedit` session has started.

```
(advice-add 'iedit-mode :after (lambda (&optional ignore)
                                (when iedit-mode
                                  (minibuffer-message "iedit session started. Press C-; to end.")))
```

Refer to version published in `unpacked.el`

[lispy: short and sweet LISP editing](#)

multi-line: multi-line everything from function invocations and definitions to array and map literals in a wide variety of languages

multiple-cursors.el: Multiple cursors

smartparens: Minor mode that deals with parens pairs and tries to be smart about it

General

- [Libraries](#)
 - [Common Lisp Extensions \(cl-lib\)](#)
 - [dash.el](#)
 - [loop.el](#): friendly imperative loop structures
 - [subr-x](#)
- [Tools](#)
 - [chemacs](#): Emacs profile switcher
 - [el2markdown](#): Convert Emacs Lisp comments to Markdown
 - [multicolumn](#): Multiple side-by-side windows support
 - [lentic](#): Create views of the same content in two buffers
 - [suggest.el](#): discover elisp functions that do what you want
 - [Byte-compile and load directory](#)
 - [emacs-lisp-macroreplace](#)
- [Tutorials](#)
 - [Wilfred Hughes walks through a function from Lispy](#)

Libraries

Common Lisp Extensions (cl-lib)

This is the built-in `cl-lib` package which implements Common Lisp functions and control structures for Emacs Lisp.

dash.el

Dash is a powerful general-purpose library that provides many useful functions and macros.

loop.el: friendly imperative loop structures

Emacs Lisp is missing loop structures familiar to users of newer languages. This library adds a selection of popular loop structures as well as `break` and `continue`.

subr-x

Less commonly used functions that complement basic APIs, often implemented in C code (like hash-tables and strings), and are not eligible for inclusion in `subr.el`.

This is a built-in package that provides several useful functions and macros, such as `thread-first / last`, `if-let / when-let`, hash-table functions, and string functions. It's easy to forget about this, since:

Do not document these functions in the lispref. <http://lists.gnu.org/archive/html/emacs-devel/2014-01/msg01006.html>

Tools

[chemacs: Emacs profile switcher](#)

This package may be especially helpful for developing in one's own environment and testing in another, like default Emacs, Spacemacs, etc.

Chemacs is an Emacs profile switcher, it makes it easy to run multiple Emacs configurations side by side. Think of it as a bootloader for Emacs.

Emacs configuration is either kept in a `~/.emacs` file or, more commonly, in a `~/.emacs.d` directory. These paths are hard-coded. If you want to try out someone else's configuration, or run different distributions like Prelude or Spacemacs, then you either need to swap out `~/.emacs.d`, or run Emacs with a different `$HOME` directory set. This last approach is quite common, but has some real drawbacks, since now packages will no longer know where your actual home directory is.

All of these makes trying out different Emacs configurations and distributions needlessly cumbersome. Various approaches to solving this have been floated over the years. There's an Emacs patch around that adds an extra command line option, and various examples of how to add a command line option in userspace from Emacs Lisp.

Chemacs tries to implement this idea in a user-friendly way, taking care of the various edge cases and use cases that come up.

[el2markdown: Convert Emacs Lisp comments to MarkDown](#)

[multicolumn: Multiple side-by-side windows support](#)

[lentic: Create views of the same content in two buffers](#)

[suggest.el: discover elisp functions that do what you want](#)

Byte-compile and load directory

Byte-compile and load all elisp files in `DIRECTORY`. Interactively, directory defaults to `default-directory` and asks for confirmation.

```
;;;###autoload
(defun epdh/byte-compile-and-load-directory (directory)
  "Byte-compile and load all elisp files in DIRECTORY.
Interactively, directory defaults to `default-directory' and asks
for confirmation."
  (interactive (list default-directory))
  (if (or (not (called-interactively-p))
        (yes-or-no-p (format "Compile and load all files in %s?" directory)))
      ;; Not sure if binding `load-path' is necessary.
      (let* ((load-path (cons directory load-path))
             (files (directory-files directory 't (rx ".el" eos))))
        (doalist (file files)
          (byte-compile-file file 'load))))))
```

emacs-lisp-macroreplace

Replace macro form before or after point with its expansion.

```
;;;###autoload
(defun epdh/emacs-lisp-macroreplace ()
  "Replace macro form before or after point with its expansion."
  (interactive)
  (if-let* ((beg (point))
            (end t))
      (form (or (ignore-errors
                  (save-excursion
                    (prog1 (read (current-buffer))
                      (setq end (point)))))
                (ignore-errors
                  (forward-sexp -1)
                  (setq beg (point))
                  (prog1 (read (current-buffer))
                    (setq end (point)))))
              (expansion (macroexpand-all form)))
        (setf (buffer-substring beg end) (pp-to-string expansion))
        (user-error "Unable to expand")))
```

Tutorials

[Wilfred Hughes walks through a function from Lispy](#)

Wilfred's walkthrough is helpful for learning how to study an Elisp function and determine what it does and how. He also shows the use of `trace-function`.

Highlighting / font-locking

- [Packages](#)
 - [lisp-extra-font-lock: Highlight bound variables and quoted expressions in lisp](#)
- [Tools](#)

- [face-explorer](#): Library and tools for faces and text properties
- [faceup](#): Regression test system for font-lock keywords
- [font-lock-profiler](#): Coverage and timing tool for font-lock keywords
- [font-lock-regression-suite](#): Regression test suite for font-lock keywords of Emacs standard modes
- [font-lock-studio](#): Debugger for Font Lock keywords
- [highlight-refontification](#): Visualize how font-lock refontifies a buffer

Packages

Packages that do highlighting/font-locking.

[lisp-extra-font-lock](#): Highlight bound variables and quoted expressions in lisp

Tools

Tools for developing highlighting/font-locking packages.

[face-explorer](#): Library and tools for faces and text properties

[faceup](#): Regression test system for font-lock keywords

[font-lock-profiler](#): Coverage and timing tool for font-lock keywords

[font-lock-regression-suite](#): Regression test suite for font-lock keywords of Emacs standard modes

[font-lock-studio](#): Debugger for Font Lock keywords

[highlight-refontification](#): Visualize how font-lock refontifies a buffer

Multiprocessing (generators, threads)

- [Articles](#)
- [Libraries](#)
 - [emacs-ai](#): async/await for Emacs Lisp
- [Manual](#)

Articles

[Emacs 26 Brings Generators and Threads « null program](#)

:archive.today: <http://archive.today/lrane>

Chris Wellons explains the new generators and threads that Emacs 26 provides. He also shows an example of writing a `cl-case` form that uses the new `switch` jump table opcode in Emacs 26.

[Turning Asynchronous into Synchronous in Elisp « null program](#)

:archive.today: <http://archive.today/AfL0y>

[Asynchronous Requests from Emacs Dynamic Modules « null program](#)

:archive.today: <http://archive.today/ZS6pU>

Libraries

[emacs-ai](#): async/await for Emacs Lisp

`ai` is to Emacs Lisp as `asyncio` is to Python. This package builds upon Emacs 25 generators to provide functions that pause while they wait on asynchronous events. They do not block any thread while paused.

Manual

[GNU Emacs Lisp Reference Manual: Generators](#)

[GNU Emacs Lisp Reference Manual: Threads](#)

Networking

HTTP

Libraries

For simple use cases, and some more complex ones, the built-in `url` library should be sufficient. Libraries that use `curl`, such as `request`, can provide better performance and more flexibility. However, in this author's experience, both of those tools, while mostly reliable, tend to have some obscure bugs that can occasionally be problematic.

- [elfeed-curl](#)
- [emacs-curl](#): CURL wrapper
- [grapnel](#): HTTP request lib built on curl with flexible callback dispatch
- [Request.el – Easy HTTP requests](#)
- [url](#)

[elfeed-curl](#)

Not a standalone package, but part of [Elfeed](#). A solid, well-designed library, but purpose-built for Elfeed. Could easily be adapted to other packages or factored out as a separate package.

emacs-curl: CURL wrapper

A very simple `curl` wrapper, last updated in 2012. Not published on MELPA.

grapnel: HTTP request lib built on curl with flexible callback dispatch

A flexible, featureful `curl` wrapper, last updated in 2015.

Request.el – Easy HTTP requests

`request` is the most commonly used third-party HTTP library. It has both `curl` and `url.el` backends.

url

`url` is included with Emacs and used by a variety of packages.

Packaging

- [Articles](#)
 - [Good Style in modern Emacs Packages](#)
- [Best practices](#)
 - [Autoloads](#)
 - [Integration with other packages](#)
 - [Lexical binding](#)
 - [Template](#)
 - [Readme](#)
 - [Version numbers](#)
- [Libraries](#)
- [Reference](#)
 - [Package headers and structure](#)
- [Tools](#)
 - [Building / Testing](#)
 - [Package installation/management](#)

Articles

[Good Style in modern Emacs Packages](#)

Best practices

Autoloads

Autoloading macro-generated functions

This may actually be a bug, or at least an unanswered question.

[How to use autoload cookies for custom defun-like macros? : emacs:](#)

Say I have a macro `deffoo` that expands to some custom kind of `defun`, and I want to use an autoload cookie to autoload the result. According to the manual,

```
;;;###autoload (deffoo bar ...)
```

copies the entire form to `autoloads.el`, and something like

```
;;;###autoload (autoload 'bar "this-file") (deffoo bar ...)
```

should be used instead. What confuses me is [this StackOverflow comment](#) by who appears to be Stefan Monnier, saying that Emacs *should* expand the macro before generating the autoload, and that it's probably a bug when this does not happen.

Can anyone clear up what the intended behaviour is?

[2018-01-15 Mon 03:37] The correct way to do this is documented in [this bug report](#).

Articles

[Autoloads in Emacs Lisp | Sebastian Wiesner](#)

:archive.today: <http://archive.today/UZHhS>

Integration with other packages

Optional support

Sometimes you want your package to integrate with other packages, but you don't want to require users to install those other packages. For example, you might want your package to work with Helm, Ivy, or the built-in Emacs `completing-read`, but you don't want to declare a dependency on and `require` Helm or Ivy, which would force users to install them to use your package.

The best way to handle this is with the `=with-eval-after-load=` macro. The [Emacs manual](#) has a page on it, and [this StackOverflow question](#) has some more info. You can also see an [example](#), which also [uses `=declare-function=`](#) to prevent byte-compiler warnings. Note as well that, according to [this StackOverflow comment](#), when a function call is guarded by `fboundp`, it's not necessary to use `declare-function` to avoid a warning.

Lexical binding

You should always use lexical binding by setting the header in the first line of the file:

```
;; filename.el --- File description  -*- lexical-binding: t; -*-
```

Articles

[Emacs Lisp lexical binding gotchas and related best practices | Yoo Box](#)

:archive.today: <http://archive.today/OnfB4>

[elisp - Why is 'let' faster with lexical scope? - Emacs Stack Exchange](#)

:archive.today: <http://archive.today/LUtzZ>

Sebastian Wiesner provides a detailed explanation.

[EmacsWiki: Dynamic Binding Vs Lexical Binding](#)

:archive.today: <http://archive.today/2VtOU>

A lot of good examples and discussion.

Some Performance Advantages of Lexical Scope « null program

Template

When you make a new package, the `auto-insert` command will insert a set of standard package headers for you. However, here is a more comprehensive template you can use:

```
;; package-name.el --- Package description (don't include the word "Emacs")  -*- lexical-binding: t
;;
;; Copyright (C) 2017 First Last
;;
;; Author: First Last <name@example.com>
;; URL: https://example.com/package-name.el
;; Version: 0.1-pre
;; Package-Requires: ((emacs "25.2"))
;; Keywords: something
;;
;; This file is not part of GNU Emacs.
;;
;; This program is free software; you can redistribute it and/or modify
;; it under the terms of the GNU General Public License as published by
;; the Free Software Foundation, either version 3 of the License, or
;; (at your option) any later version.
;;
;; This program is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY; without even the implied warranty of
;; MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
;; GNU General Public License for more details.
;;
;; You should have received a copy of the GNU General Public License
;; along with this program. If not, see <http://www.gnu.org/licenses/>.
;;
;;; Commentary:
;;
;; This package allows flanges to be easily frobnicated.
;;
;;; Installation
;;
;;; MELPA
;;
;; If you installed from MELPA, you're done.
;;
;;; Manual
;;
;; Install these required packages:
;;
;; + foo
;; + bar
;;
;; Then put this file in your load-path, and put this in your init
;; file:
;;
;; (require 'package-name)
;;
;;; Usage
;;
;; Run one of these commands:
;;
;; `package-name-command': Frobnicate the flange.
;;
;;; Tips
;;
;; + You can customize settings in the `package-name' group.
;;
;;; Credits
;;
;; This package would not have been possible without the following
;; packages: foo[1], which showed me how to bifurcate, and bar[2],
;; which takes care of flanges.
;;
```



```

- [[#installation][Installation]]
- [[#usage][Usage]]
- [[#changelog][Changelog]]
- [[#credits][Credits]]
- [[#development][Development]]
- [[#license][License]]
:END:

* Installation
:PROPERTIES:
:TOC:      :depth 0
:END:

** MELPA

If you installed from MELPA, you're done. Just run one of the commands below.

** Manual

Install these required packages:

+ =foo=
+ =bar=

Then put this file in your load-path, and put this in your init file:

#+BEGIN_SRC elisp
(require 'package-name)
#+END_SRC

* Usage
:PROPERTIES:
:TOC:      :depth 0
:END:

Run one of these commands:

+ =package-name-command=: Frobnicate the flange.

** Tips

+ You can customize settings in the =package-name= group.

* Changelog
:PROPERTIES:
:TOC:      :depth 0
:END:

** 1.1.0

*Additions*
+ Add command =package-name-debarnacle= to de-barnacle the hull.

*Changes*
+ Command =package-name-anchor= now takes an argument, =weigh= or =let-go=.

*Internal*
+ Rewrote input parsing.
+ Factored out anchor-weighing.

** 1.0.1

*Fixes*
+ Ensure anchor is secure before returning from =package-name-anchor=.

** 1.0.0

Initial release.

* Credits

This package would not have been possible without the following packages: [[https://example.com/fo

* Development

Bug reports, feature requests, suggestions – /oh my/!

* License

GPLV3

# Local Variables:
# eval: (require 'org-make-toc)
# before-save-hook: org-make-toc
# org-export-with-properties: ()
# org-export-with-title: t
# End:

```

Version numbers

Version numbers which are valid in Emacs are those accepted by the function `version-to-list`, which uses the variables `version-separator` and `version-regexp-alist`. See their documentation for specific, up-to-date information. `version-to-list`'s documentation (as of Emacs 26.1) is reproduced here for convenience:

The version syntax is given by the following EBNF:

```
VERSION ::= NUMBER ( SEPARATOR NUMBER )*.
```

```
NUMBER ::= (0|1|2|3|4|5|6|7|8|9)+.
```

```
SEPARATOR ::= 'version-separator' (which see)
| 'version-regexp-alist' (which see).
```

The NUMBER part is optional if SEPARATOR is a match for an element in 'version-regexp-alist'.

Examples of valid version syntax:

```
1.0pre2 1.0.7.5 22.8beta3 0.9alpha1 6.9.30Beta 2.4.snapshot .5
```

Examples of invalid version syntax:

```
1.0prepre2 1.0..7.5 22.8X3 alpha3.2
```

Examples of version conversion:

Version String Version as a List of Integers

```
"5" (0 5)
"0.9 alpha" (0 9 -3)
"0.9Alpha1" (0 9 -3 1)
"0.9snapshot" (0 9 -4)
"1.0-git" (1 0 -4)
"1.0.7.5" (1 0 7 5)
"1.0.cvs" (1 0 -4)
"1.0PRE2" (1 0 -1 2)
"1.0pre2" (1 0 -1 2)
"22.8 Beta3" (22 8 -2 3)
"22.8beta3" (22 8 -2 3)
```

Libraries

`lisp-mnt.el` (`lm`)

This library includes functions helpful for working with and verifying the format of Emacs Lisp package files, including headers, commentary, etc. It's easy to overlook and hard to re-discover this package because of its `lm` symbol prefix. It's listed here because your editor keeps forgetting what it's called.

Reference

Package headers and structure

The [Emacs manual](#) gives this example (I've added the lexical-binding part). Also see [template](#).

```
;;; superfrobnicator.el --- Frobnicate and bifurcate flanges -*- lexical-binding: t; -*-

;; Copyright (C) 2011 Free Software Foundation, Inc.

;; Author: J. R. Hacker <jrh@example.com>
;; Version: 1.3
;; Package-Requires: ((flange "1.0"))
;; Keywords: multimedia, frobnicate
;; URL: http://example.com/jrhacker/superfrobnicate

...

;;; Commentary:

;; This package provides a minor mode to frobnicate and/or
;; bifurcate any flanges you desire. To activate it, just type
...

;;;###autoload
(define-minor-mode superfrobnicator-mode
...

```

Tools

- [Building / Testing](#)
 - [cask](#): Project management tool for Emacs
 - [eldev](#): Elisp Development Tool
 - [emacs-package-checker](#): Check Emacs Lisp packages in a clean environment
 - [emake.el](#): Test Elisp without the hoops
 - [make1](#): A makefile to facilitate checking Emacs packages
 - [makem.sh](#): Makefile-like script for building and testing packages
- [Package installation/management](#)
 - [straight.el](#): Next-generation, purely functional package manager for the Emacs hacker
 - [use-package](#): A use-package declaration for simplifying your .emacs
 - [paradox](#): modernizing Emacs' Package Menu. With package ratings, usage statistics, customizability, and more.
 - [el-get](#): Manage the external elisp bits and pieces upon which you depend!

Building / Testing

Tools for building and testing packages, especially from scripts or Makefiles.

[cask](#): Project management tool for Emacs

Cask is a project management tool for Emacs that helps automate the package development cycle; development, dependencies, testing, building, packaging and more.

eldev: [Elisp Development Tool](#)

Eldev (Elisp Development Tool) is an Emacs-based build tool, targeted solely at Elisp projects. It is an alternative to Cask. Unlike Cask, Eldev itself is fully written in Elisp and its configuration files are also Elisp programs. If you are familiar with Java world, Cask can be seen as a parallel to Maven — it uses project description, while Eldev is sort of a parallel to Gradle — its configuration is a program on its own.

emacs-package-checker: [Check Emacs Lisp packages in a clean environment](#)

Emacs Package Checker lets you quickly configure typical linters (i.e. package-lint, byte-compile, and checkdoc) for your Emacs package.

There are existing solutions in this field like emake.el and makel. Emacs Package Checker is not any more capable than those existing solutions, but it is based on Nix package manager and runs tests in a pure, sandboxed environment. This is useful for testing Emacs packages on local machines.

emake.el: [Test Elisp without the hoops](#)

Test Elisp with services like Travis CI without the fuss of Cask – just you, your project, and (Emacs-)Make.

Things EMake does:

- parses, installs, and runs tests for your package
- provides all the power of Elisp to extend its capabilities on-demand

Things EMake will never do (or 'reasons you may still need Cask'):

- manage your development environment or provide tools to do so
- provide 'bundler-like' exec abilities (this includes Cask's emacs and eval commands)

makel: [A makefile to facilitate checking Emacs packages](#)

makel is a project consisting of a Makefile (`makel.mk`) that Emacs package authors can use to facilitate quality checking (linting and tests). The Makefile can be used both locally on the developer machine and remotely on a continuous integration machine.

makem.sh: [Makefile-like script for building and testing packages](#)

makem.sh is a script helps to build, lint, and test Emacs Lisp packages. It aims to make linting and testing as simple as possible without requiring per-package configuration.

It works similarly to a Makefile in that "rules" are called to perform actions such as byte-compiling, linting, testing, etc.

Source and test files are discovered automatically from the project's Git repo, and package dependencies within them are parsed automatically.

Output is simple: by default, there is no output unless errors occur. With increasing verbosity levels, more detail gives positive feedback. Output is colored by default to make reading easy.

The script can run Emacs with the developer's local Emacs configuration, or with a clean, "sandbox" configuration that can be optionally removed afterward. This is especially helpful when upstream dependencies may have released new versions that differ from those installed in the developer's personal configuration.

Package installation/management

straight.el: [Next-generation, purely functional package manager for the Emacs hacker](#)

- State "TODO" from [2018-07-29 Sun 13:11]

use-package: [A use-package declaration for simplifying your .emacs](#)

- State "TODO" from [2018-07-29 Sun 13:11]

Developed by the current maintainer of Emacs, himself, John Wiegley.

paradox: [modernizing Emacs' Package Menu. With package ratings, usage statistics, customizability, and more.](#)

el-get: [Manage the external elisp bits and pieces upon which you depend!](#)

Pattern matching

- [Articles](#)
- [Libraries](#)
 - [dash.el](#)
 - [pcase](#)
 - [shadchen-el](#)
- [Tools](#)
 - [let-alist](#)
 - [with-dict](#), [with-plist-vals](#)

Articles

Pattern Matching in Emacs Lisp – Wilfred Hughes::Blog

:archive.today: <http://archive.today/J4DqY>

Pattern matching is invaluable in elisp. Lists are ubiquitous, and a small amount of pattern matching can often replace a ton of verbose list fiddling.

Since this is Lisp, we have lots of choices! In this post, we'll compare [cl.el](#), [pcase.el](#), [dash.el](#), and [shadchen](#), so you can choose the best fit for your project. We'll look at the most common use cases, and end with some recommendations.

For the sake of this post, we'll consider both pattern matching and destructuring, as they're closely related concepts.

A callable plist data structure for Emacs

:archive.today: <http://archive.today/vmlTX>

John Kitchin demonstrates some macros that make it easy to access plist values.

Libraries

[dash.el](#)

Dash is a powerful library, and one of its features is powerful destructuring with its `-let` macro, and several others that work the same way.

[pcase](#)

`pcase` is built-in to Emacs. Its syntax can be confusing, but it is very powerful.

Articles

[Emacs: Pattern Matching with `pcase` - Lost in Technopolis](#)

:archive.today: <http://archive.today/FAzd8>

This tutorial by John Wiegley is a great introduction to `pcase`.

[EmacsWiki: Pattern Matching](#)

There are *lots* of examples here.

Nic Ferrier, [Using Polymorphism as a Lisp refactoring tool](#)

:archive.today: <http://archive.today/OY3Md>

Examples

`dash`

[2018-07-27 Fri 23:29] Dash has new abilities, including `-setq`, and destructuring plists with implied variable names (i.e. just the keys can be specified, reducing repetition).

`pcase-let`

This example shows the use of `pcase-let*` to destructure and bind a nested alist:

```
(let ((alphabets (a-list 'English (a-list 'first "a"
                                          'second "b")
                          'Greek (a-list 'first "α"
                                          'second "β")))))
  (pcase-let* (((map English) alphabets)
              ((map ('first letter) second) English))
    (list letter second))) ;; => ("a" "b")
```

[shadchen-el](#)

A powerful, Racket-style pattern-matching library.

Tools

`let-alist`

`let-alist` is the best thing to happen to associative lists since the invention of the cons cell. This little macro lets you easily access the contents of an alist, concisely and efficiently, without having to specify them preemptively. It comes built-in with 25.1, and is also available on GNU Elpa for older Emacsen.

Example:

```
(defun sx-question-list--print-info (question-data)
  "DOC"
  (let-alist question-data
    (list
      question-data
      (vector
        (int-to-string .score)
        (int-to-string .answer_count)
        .title " "
        .owner.display_name
        .last_activity_date sx-question-list-ago-string
        " " .tags))))
```

Articles

[New on Elpa and in Emacs 25.1: `let-alist` · Endless Parentheses](#)

:archive.today: <http://archive.today/2wNFm>

Here Artur introduces the macro and gives examples.

with-dict, with-plist-vals

Courtesy of John Kitchin:[fn:1:Copyright by John Kitchin, licensed under a Creative Commons Attribution-ShareAlike 4.0 International License.]

```
(defmacro with-dict (key-vals &rest body)
  "A context-manager for a plist where each key is a callable
  function that returns the value."
  (declare (indent 1))
  (let* ((g (if (symbolp key-vals)
                (symbol-value key-vals)
                key-vals))
        (keys (-slice g 0 nil 2)))
    `(labels ,(loop for key in keys
                    collect
                    (list key '() `(plist-get ',g ,key)))
      ,@body)))

;; Used as:

(with-dict (:a 1 :b 'some-symbol :c 3)
  (:b))

(let ((d '(:key1 1 :key2 some-other-symbol :key3 3)))
  (with-dict d
    (format "We got %s" (:key2))))
```

And:

```
(defmacro with-plist-vals (plist &rest body)
  "Bind the values of a plist to variables with the name of the keys."
  (declare (indent 1))
  `(let ,(loop for key in (-slice plist 0 nil 2)
              for val in (-slice plist 1 nil 2)
              collect (list (intern
                            (substring (symbol-name key) 1))
                            val))
    ,@body))

;; Used like:

(with-plist-vals (:a 4 :b 6)
  (* 2 a))
```

Processes (incl. IPC, RPC)

Including inter-process communication (IPC) and remote procedure calls (RPC).

Libraries

debase: D-Bus<->EIEIO bridge

D-Bus is an IPC system which is ubiquitous on Linux, and (in this author's opinion) not very good. Emacs has bindings for interfacing with it (see the former point), which are annoying to use (see the latter point).

These days, numerous common system management tasks are implemented as D-Bus services rather than traditional *nix commands, and many of the command-line tools themselves are now front-ends which communicate via D-Bus. Mounting and unmounting disks, monitoring battery status, controlling display brightness, connecting to wireless networks and more are now handled with D-Bus services.

It makes no sense to shell out to the tools when one could interact with them directly via D-Bus, if only it was less annoying to do so.

Debase frees you from writing repetitive, annoying boilerplate code to drive D-Bus services by throwing another pile of abstraction at the problem, in the form of unreadably dense, macro-heavy, profoundly cursed Lisp.

Optimization

Including benchmarking, byte-compilation, profiling, etc.

- [Articles](#)
 - [Efficient Alias of a Built-In Emacs Lisp Function](#) « null program
 - [Emacs Byte-code Internals](#)
 - [Faster Elfeed Search Through JIT Byte-code Compilation](#)
 - [How to Write Fast\(er\) Emacs Lisp](#) « null program
 - [Some Performance Advantages of Lexical Scope](#) « null program
 - [What's in an Emacs Lambda](#) « null program
- [Reference](#)
 - [elisp-bytecode: Let's document Emacs Lisp Bytecode \(Lisp Assembly Program\) instructions](#)
- [Tools](#)
 - [Benchmarking](#)
 - [bench macro](#)
 - [bench-multi macros](#)
 - [bench-multi-lexical](#)
 - [bench-dynamic-vs-lexical-binding](#)
 - [bench-multi-lets](#)
 - [Profiling](#)

- [elp-profile](#)
- [etrace: Emacs Lisp Latency Tracing for the Chromium Catapult Trace Event Format](#)

Articles

- [Efficient Alias of a Built-In Emacs Lisp Function « null program](#)
- [Emacs Byte-code Internals](#)
- [Faster Elfeed Search Through JIT Byte-code Compilation](#)
- [How to Write Fast\(er\) Emacs Lisp « null program](#)
- [Some Performance Advantages of Lexical Scope « null program](#)
- [What's in an Emacs Lambda « null program](#)

Efficient Alias of a Built-In Emacs Lisp Function « null program

:archive.is: <http://archive.today/wip/pdlQH>

Chris Wellons compares the use of `defalias` and `defsubst` and how they are optimized by the byte-compiler.

Emacs Byte-code Internals

:archive.is: <http://archive.today/DFx3S>

Byte-code compilation is an underdocumented — and in the case of the recent lexical binding updates, undocumented — part of Emacs. Most users know that Elisp is usually compiled into a byte-code saved to `.elc` files, and that byte-code loads and runs faster than uncompiled Elisp. That's all users really need to know, and the *GNU Emacs Lisp Reference Manual* specifically discourages poking around too much.

People do not write byte-code; that job is left to the byte compiler. But we provide a disassembler to satisfy a cat-like curiosity.

Screw that! What if I want to handcraft some byte-code myself? :-) The purpose of this article is to introduce the internals of Elisp byte-code interpreter. I will explain how it works, why lexically scoped code is faster, and demonstrate writing some byte-code by hand.

Faster Elfeed Search Through JIT Byte-code Compilation

Chris Wellons shows how he substantially improved Elfeed's search by byte-compiling functions at runtime and avoiding function call overhead. He also demonstrates some simple benchmarking tools he wrote in the process.

How to Write Fast(er) Emacs Lisp « null program

:archive.today: <http://archive.today/xe0Js>

Chris Wellons explains five ways to write faster Emacs Lisp code.

Some Performance Advantages of Lexical Scope « null program

:archive.today: <http://archive.today/xm5zq>

I recently had a discussion with Xah Lee about lexical scope in Emacs Lisp. The topic was *why* lexical-binding exists at a file-level when there was already lexical-let (from `cl-lib`), prompted by my previous article on JIT byte-code compilation. The specific context is Emacs Lisp, but these concepts apply to language design in general.

What's in an Emacs Lambda « null program

:archive.today: <http://archive.today/ppluJ>

Chris explains how lambdas work with regard to byte-compilation and lexical binding.

Reference

elisp-bytecode: Let's document Emacs Lisp Bytecode (Lisp Assembly Program) instructions

Rocky Bernstein's project to document the Elisp bytecode format (aka LAP, or Lisp Assembly Program).

Tools

Benchmarking

bench macro

From Phil Lord's `m-buffer-el`:

```
;;###autoload
(cl-defmacro bench (&optional (times 100000) &rest body)
  "Call 'benchmark-run-compiled' on BODY with TIMES iterations, returning list suitable for Org source
  Garbage is collected before calling 'benchmark-run-compiled' to
  avoid counting existing garbage which needs collection."
  (declare (indent defun))
  `(progn
    (garbage-collect)
    (list ("Total runtime" "# of GCs" "Total GC runtime")
      'hline
      (benchmark-run-compiled ,times
        (progn
          ,@body))))))
```

Used like this:

```
(bench 1000000
  (cons 'time (current-time)))
```

When called from an Org source block, it gives output like this:

Total runtime	# of GCs	Total GC runtime
1.657838266	3	1.4723854609999876

bench-multi macros

These macros make comparing multiple forms easy:

```
;;###autoload
(cl-defmacro bench-multi (&key (times 1) forms ensure-equal raw)
  "Return Org table as a list with benchmark results for FORMS.
  Runs FORMS with 'benchmark-run-compiled' for TIMES iterations.

  When ENSURE-EQUAL is non-nil, the results of FORMS are compared,
  and an error is raised if they aren't 'equal'. If the results are
  sequences, the difference between them is shown with
  'seq-difference'.

  When RAW is non-nil, the raw results from
  'benchmark-run-compiled' are returned instead of an Org table
  list.

  If the first element of a form is a string, it's used as the
  form's description in the bench-multi-results; otherwise, forms
  are numbered from 0.

  Before each form is run, 'garbage-collect' is called."
  ;; MAYBE: Since 'bench-multi-lexical' byte-compile the file, I'm not sure if
  ;; 'benchmark-run-compiled' is necessary over 'benchmark-run', or if it matters.
  (declare (indent defun))
  (let* ((keys (gensym "keys"))
         (result-times (gensym "result-times"))
         (header '(("Form" "x fastest" "Total runtime" "# of GCs" "Total GC runtime")
                   hline))
         ;; Copy forms so that a subsequent call of the macro will get the original forms.
         (forms (copy-list forms))
         (descriptions (cl-loop for form in forms
                                for i from 0
                                collect (if (stringp (car form))
                                             (progn1 (car form)
                                                       (setf (nth i forms) (cadr (nth i forms))))
                                             i))))
    `(unwind-protect
       (progn
        (defvar bench-multi-results nil)
        (let* ((bench-multi-results (make-hash-table))
               (,result-times (sort (list ,@(cl-loop for form in forms
                                                       for i from 0
                                                       for description = (nth i descriptions)
                                                       collect `(progn
                                                                    (garbage-collect)
                                                                    (cons ,description
                                                                    (benchmark-run-compiled ,ti
                                                                    ,if ensure-equal
                                                                    (puthash ,descripti
                                                                    form))))))
                                   (lambda (a b)
                                     (< (second a) (second b))))))
          ,when ensure-equal
          `(cl-loop with ,keys = (hash-table-keys bench-multi-results)
                    for i from 0 to (- (length ,keys) 2)
                    unless (equal (gethash (nth i ,keys) bench-multi-results)
                                  (gethash (nth (1+ i) ,keys) bench-multi-results))
                    do (if (sequencep (gethash (car (hash-table-keys bench-multi-results)) bench-multi-results))
                           (let* ((k1) (k2))
                              ;; If the difference in one order is nil, try in other order
                              (difference (or (setq k1 (nth i ,keys)
                                                    k2 (nth (1+ i) ,keys)
                                                    difference (seq-difference (gethash k1
                                                                                          (gethash k2
                                                                                          (setq k1 (nth (1+ i) ,keys)
                                                                                          k2 (nth i ,keys)
                                                                                          difference (seq-difference (gethash k1
                                                                                          (gethash k2
                                                                                          (setq k1 (nth (1+ i) ,keys)
                                                                                          k2 (nth i ,keys)
                                                                                          difference (seq-difference (gethash k1
                                                                                          (gethash k2
                                                                                          (user-error "Forms' bench-multi-results not equal: difference (%s
                                                                                          k1 k2 difference))
                                                                                          ;; Not a sequence
                                                                                          (user-error "Forms' bench-multi-results not equal: %s:%S %s:%S"
                                                                                          (nth i ,keys) (nth (1+ i) ,keys)
                                                                                          (gethash (nth i ,keys) bench-multi-results)
                                                                                          (gethash (nth (1+ i) ,keys) bench-multi-results))))))
                              ;; Add factors to times and return table
                              (if ,raw
                                  ,result-times
                                  (append ',header
                                  (bench-multi-process-results ,result-times))))))
          (unintern 'bench-multi-results nil))))
        (defun bench-multi-process-results (results)
          "Return sorted RESULTS with factors added."
          (setq results (sort results (-on #'< #'second)))
          (cl-loop with length = (length results)
                    for i from 0 below length
                    for description = (car (nth i results))
                    for factor = (pcase i
                                  (0 "fastest")
                                  (_ (format "%.2f" (/ (second (nth i results))
                                                         (second (nth 0 results))))))
                    collect (append (list description factor)
```

```
(list (format "%.6f" (second (nth i results)))
      (third (nth i results))
      (if (> (fourth (nth i results)) 0)
          (format "%.6f" (fourth (nth i results)))
          0))))))
```

Used like:

```
(bench-multi
 :forms (("org-map-entries" (sort (org-map-entries (lambda ()
                                                    (nth 4 (org-heading-components)))
                                                    "/+MAYBE" 'agenda)
                                     #'string<))
          ("regexp" (sort (-flatten
                           (-non-nil
                            (mapcar (lambda (file)
                                      (let ((case-fold-search t))
                                        (with-current-buffer (find-buffer-visiting file)
                                          (org-with-wide-buffer
                                           (goto-char (point-min))
                                           (cl-loop with regexp = (format org-heading-keyword-regexp
                                                                           while (re-search-forward regexp nil t)
                                                                           collect (nth 4 (org-heading-components))))))
                                      (org-agenda-files))))
                                     #'string<))))))
```

#+RESULTS[3316dc4375a3b162e32790bb7e72d715d7f756fb]:

Form	x fastest	Total runtime	# of GCs	Total GC runtime
regexp	fastest	0.022259	0	0
org-map-entries	168.03	3.740340	0	0

It can also help catch bugs by ensuring that each form returns the same results. For example, the benchmark above contains a subtle bug: because `case-fold-search` in the `regexp` form is non-nil, the `regexp` is compared case-insensitively, so it matches Org headings which start with `Maybe` rather than only ones which start with `MAYBE`. Using the `:ensure-equal t` argument to `bench-multi` compares the results and raises an error showing the difference between the two sequences the forms evaluate to:

```
(bench-multi :ensure-equal t
 :forms (("org-map-entries" (sort (org-map-entries (lambda ()
                                                    (nth 4 (org-heading-components)))
                                                    "/+MAYBE" 'agenda)
                                     #'string<))
          ("regexp" (sort (-flatten
                           (-non-nil
                            (mapcar (lambda (file)
                                      (let ((case-fold-search t))
                                        (with-current-buffer (find-buffer-visiting file)
                                          (org-with-wide-buffer
                                           (goto-char (point-min))
                                           (cl-loop with regexp = (format org-heading-keyword-regexp
                                                                           while (re-search-forward regexp nil t)
                                                                           collect (nth 4 (org-heading-components))))))
                                      (org-agenda-files))))
                                     #'string<))))))
```

user-error: Forms' results not equal: difference (regexp - org-map-entries): ("Maybe this is not the

Fixing the error, by setting `case-fold-search` to `nil`, not only makes the forms give the same result but, in this case, doubles the performance of the faster form:

```
(bench-multi :ensure-equal t
 :forms (("org-map-entries" (sort (org-map-entries (lambda ()
                                                    (nth 4 (org-heading-components)))
                                                    "/+MAYBE" 'agenda)
                                     #'string<))
          ("regexp" (sort (-flatten
                           (-non-nil
                            (mapcar (lambda (file)
                                      (let ((case-fold-search nil))
                                        (with-current-buffer (find-buffer-visiting file)
                                          (org-with-wide-buffer
                                           (goto-char (point-min))
                                           (cl-loop with regexp = (format org-heading-keyword-regexp
                                                                           while (re-search-forward regexp nil t)
                                                                           collect (nth 4 (org-heading-components))))))
                                      (org-agenda-files))))
                                     #'string<))))))
```

#+RESULTS[773b94ff27f73dcfb694429054710a581b7bec5]:

Form	x fastest	Total runtime	# of GCs	Total GC runtime
regexp	fastest	0.011578	0	0
org-map-entries	313.65	3.631561	0	0

So this macro showed which code is faster and helped catch a subtle bug.

bench-multi-lexical

To evaluate forms with lexical binding enabled, use this macro:

```
;;###autoload
(cl-defmacro bench-multi-lexical (&key (times 1) forms ensure-equal raw)
  "Return Org table as a list with benchmark results for FORMS.
  Runs FORMS from a byte-compiled temp file with `lexical-binding'
  enabled, using `bench-multi', which see.

  Afterward, the temp file is deleted and the function used to run
  the benchmark is uninterned."
  (declare (indent defun))
  `(let* ((temp-file (concat (make-temp-file "bench-multi-lexical-" ) ".el"))
         (fn (gensym "bench-multi-lexical-run-")))
    (with-temp-file temp-file
      (insert ;; -*- lexical-binding: t; -*- "
              "\n\n"
              "(defvar bench-multi-results)" "\n\n"
              (format "(defun %s () (bench-multi :times %d :ensure-equal %s :raw %s :forms %S))"
                      fn ,times ,ensure-equal ,raw ',forms)))
      (unwind-protect
        (if (byte-compile-file temp-file 'load)
            (funcall (intern (symbol-name fn)))
            (user-error "Error byte-compiling and loading temp file"))
        (delete-file temp-file)
        (unintern (symbol-name fn) nil)))))
```

Used just like bench-multi:

```
(bench-multi-lexical :ensure-equal t
  :forms (("org-map-entries" (sort (org-map-entries (lambda ()
                                                    (nth 4 (org-heading-components)))
                                                    "/*MAYBE" 'agenda)
                                  #'string<))
          ("regexp" (sort (-flatten
                           (-non-nil
                             (mapcar (lambda (file)
                                       (let ((case-fold-search nil))
                                         (with-current-buffer (find-buffer-visiting file)
                                           (org-with-wide-buffer
                                             (goto-char (point-min))
                                             (cl-loop with regexp = (format org-heading-keyword-regexp
                                                                              while (re-search-forward regexp nil t)
                                                                              collect (nth 4 (org-heading-components))))))
                                       (org-agenda-files))))
                                  #'string<))))))
```

#+RESULTS[a8ffc10fa4e21eb632122657312040f139b33204]:

Form	x fastest	Total runtime	# of GCs	Total GC runtime
regexp	fastest	0.011641	0	0
org-map-entries	312.46	3.637256	0	0

bench-dynamic-vs-lexical-binding

This macro compares dynamic and lexical binding.

```
;;###autoload
(cl-defmacro bench-dynamic-vs-lexical-binding (&key (times 1) forms ensure-equal)
  "Benchmark FORMS with both dynamic and lexical binding.
  Calls `bench-multi' and `bench-multi-lexical', which see."
  (declare (indent defun))
  `(let ((dynamic (bench-multi :times ,times :ensure-equal ,ensure-equal :raw t
                              :forms ,forms))
        (lexical (bench-multi-lexical :times ,times :ensure-equal ,ensure-equal :raw t
                                       :forms ,forms)))
    (header ("Form" "x fastest" "Total runtime" "# of GCs" "Total GC runtime")))
    (cl-loop for result in-ref dynamic
      do (setf (car result) (format "Dynamic: %s" (car result))))
    (cl-loop for result in-ref lexical
      do (setf (car result) (format "Lexical: %s" (car result))))
    (append (list header)
            (list 'hline)
            (bench-multi-process-results (append dynamic lexical)))))
```

Example:

```
(bench-dynamic-vs-lexical-binding :times 1000 :ensure-equal t
  :forms (("buffer-local-value" (--filter (equal 'magit-status-mode (buffer-local-value 'major-mode
                                              (buffer-list)))
      ("with-current-buffer" (--filter (equal 'magit-status-mode (with-current-buffer it
                                                                  major-mode))
                                      (buffer-list)))))
```

#+RESULTS[73cc92a5949dd2d48f029cab9557eb6132bdf1cf]:

Form	x fastest	Total runtime	# of GCs	Total GC runtime
Lexical: buffer-local-value	fastest	0.039616	0	0

Dynamic: buffer-local-value	1.18	0.046844	0	0
Dynamic: with-current-buffer	82.07	3.251161	0	0
Lexical: with-current-buffer	82.30	3.260561	0	0

The `buffer-local-value` form improved by about 24% when using lexical binding, but the `with-current-buffer` form performs the same regardless of using lexical binding.

bench-multi-lets

This macro benchmarks multiple forms in multiple environments, which is helpful for testing code that behaves differently depending on global variables.

```
;;###autoload
(cl-defmacro bench-multi-lets (&key (times 1) lets forms ensure-equal)
  "Benchmark FORMS in each of lexical environments defined in LETS.
  LETS is a list of (\"NAME\" BINDING-FORM) forms.

  FORMS is a list of (\"NAME\" FORM) forms.

  Calls 'bench-multi-lexical', which see."
  (declare (indent defun))
  (let ((benchmarks (cl-loop for (let-name let) in lets
                             collect (list 'list let-name
                                             '(let ,let
                                                  (bench-multi-lexical :times ,times :ensure-equal ,ensu
                                                                           :forms ,forms))))))
        `(let* ((results (list ,@benchmarks))
                 (header '("Form" "x fastest" "Total runtime" "# of GCs" "Total GC runtime"))
                 (results (cl-loop for (let-name let) in results
                                   append (cl-loop for result in-ref let
                                                    do (setf (car result) (format "%s: %s" let-name (car r
                                                                                          collect result))))
                 (append (list header)
                         (list 'hline)
                         (bench-multi-process-results results))))))
```

Used like:

```
(bench-multi-lets :times 100000 :ensure-equal t
:lets ((("1" ((var "1"))
          ("12345" ((var "12345"))
                  ("1234567890" ((var "1234567890")))))
:forms ((("concat" (concat "VAR: " var))
          ("format" (format "VAR: %s" var))))
```

#+RESULTS[1e38aedd90e38a2d2e1cf00aa77f13e7da51cb3]:

Form	x fastest	Total runtime	# of GCs	Total GC runtime
1: concat	fastest	0.010552	0	0
12345: concat	1.02	0.010812	0	0
1234567890: concat	1.05	0.011071	0	0
1: format	1.51	0.015928	0	0
12345: format	1.97	0.020803	0	0
1234567890: format	2.42	0.025500	0	0

Profiling

elp-profile

Call this macro from an Org source block and you'll get a results block showing which 20 functions were called the most times, how long they took to run, etc. `prefixes` should be a list of symbols matching the prefixes of the functions you want to instrument.

```
;;###autoload
(defmacro elp-profile (times prefixes &rest body)
  (declare (indent defun))
  (let (output)
    (dolist (prefix ,prefixes)
      (elp-instrument-package (symbol-name prefix)))
    (dotimes (x ,times)
      ,@body)
    (elp-results)
    (elp-restore-all)
    (point-min)
    (forward-line 20)
    (delete-region (point) (point-max))
    (setq output (buffer-substring-no-properties (point-min) (point-max)))
    (kill-buffer)
    (delete-window)
    (let ((rows (s-lines output)))
      (append (list (list "Function" "Times called" "Total time" "Average time")
                    'hline)
              (cl-loop for row in rows
                        collect (s-split (rx (1+ space)) row 'omit-nulls))))))
```

```
;; Use like this:
(elp-profile 10 '(map search goto-char car append)
  (goto-char (point-min))
  (search-forward "something"))
```

This gives a table like:

Function	Times called	Total time	Average time
mapcar	30	0.0036004130	0.0001200137
search-forward	10	2.089...e-05	2.089...e-06
goto-char	10	6.926e-06	6.926e-07
car	13	3.956...e-06	3.043...e-07
append	1	5.96e-07	5.96e-07
mapatoms	1	0	0.0

etrace: Emacs Lisp Latency Tracing for the Chromium Catapult Trace Event Format

This package for GNU Emacs allows latency tracing to be performed on Emacs Lisp code and the results output to files using the Chromium Catapult Trace Event Format. These trace files can then be loaded into trace analysis utilities in order to generate flame graphs and other useful visualisations and analyses.

Refactoring

Tools

emacs-refactor: language-specific refactoring

Emacs Refactor (EMR) is a framework for providing language-specific refactoring in Emacs. It includes refactoring commands for a variety of languages, including elisp itself!

Regular expressions

- Articles
 - Libraries
 - pcr2el: Convert between PCRE, Emacs and rx regexp syntax
 - lex
 - Tools
 - ample-regexps.el: Compose and reuse regular expressions with ease

Articles

Exploring Emacs rx Macro

:archive.today: <http://archive.today/xPWJP>

Libraries

pcr2el: Convert between PCRE, Emacs and rx regexp syntax

lex

lex is a regular expression matching engine with syntax similar to rx . It appears to be more implemented in elisp than standard Emacs regexp tools, so it may be slower, but its additional capabilities may be useful.

Format of regexps is the same as used for `rx' and `sregex'. Additions:

- (ere RE) specify regexps using the ERE syntax.
- (inter RES...) (aka &) make a regexp that only matches if all its branches match. E.g. (inter (ere ".*a.*") (ere ".*b.*")) match any string that contain both an a and a b , in any order.
- (case-fold RES...) and (case-sensitive RES...) make a regexp that is case sensitive or not, regardless of case-fold-search.

Tools

ample-regexps.el: Compose and reuse regular expressions with ease

ample-regexps complements the built-in rx macro by flexibly defining regular expressions with reusable parts. In the following example, the define-arx macro defines three things:

- A macro url-rx , which expands to a regular expression string at compile time
- A function url-rx-to-string , which can be used at runtime
- A variable url-rx-constituents , containing form definitions to use

```
(define-arx url-rx
  '((http (seq bos (group "http") "://") )
    (https (seq bos (group "https") "://") )
    (https? (seq bos (group "http" (optional "s")) "://") )
    (protocol (seq bos (group (1+ (not (any "://")))) "://"))
    (host (group (1+ (not (any "://")))))
    (path (group "/" (1+ (not (any "?"))))))
```

```
(query (seq "?" (group (1+ (not (any "#"))))))
(fragment (seq "#" (group (1+ anything)))))
```

The `url-rx` macro can then be used to test and select parts of URLs:

```
;; Accept HTTP or HTTPS
(let ((url "http://server/path?query#fragment"))
  (when (string-match (url-rx https? host path (optional query) (optional fragment)) url)
    (list (match-string 0 url)
          (match-string 1 url)
          (match-string 2 url)
          (match-string 3 url)
          (match-string 4 url)
          (match-string 5 url)))) ;=> ("http://server/path?query#fragment" "http" "server" "/path" "

;; Only accept HTTPS, not plain HTTP
(let ((url "http://server/path?query#fragment"))
  (when (string-match (url-rx https host path (optional query) (optional fragment)) url)
    (list (match-string 0 url)))) ;=> nil

;; Accept any protocol, not just HTTP
(let ((url "ftp://server/path"))
  (when (string-match (url-rx protocol host path (optional query) (optional fragment)) url)
    (list (match-string 0 url)
          (match-string 1 url)
          (match-string 2 url)
          (match-string 3 url)
          (match-string 4 url)
          (match-string 5 url)))) ;=> ("ftp://server/path" "ftp" "server" "/path" nil nil)
```

This example shows the use of a function to expand a list of strings into a sequence:

```
(define-arg cond-assignment-rx
  '((alpha_ (regexp "[[:alpha:]]_"))
    (alnum_ (regexp "[[:alnum:]]_"))
    (ws (* blank)))
  (sym (:func (lambda (_form &rest args)
                `(seq symbol-start (or ,@args) symbol-end))))
  (cond-keyword (sym "if" "elif" "while"))
  (id (sym (+ alpha_) (* alnum_)))) ; -> cond-assignment-rx

(cond-assignment-rx cond-keyword ws id ":" id ws "=" ws id) ; -> "\\_<\\(?:elif\\|if\\|while\\)\\_>
```

Strings

- [Articles](#)
 - [Buffer-Passing Style](#)
- [Libraries](#)
 - [s.el: The long lost Emacs string manipulation library](#)
- [Tools](#)
 - [format\\$ macro](#)

Articles

Buffer-Passing Style

:archive.today: <https://archive.ph/JygiI>

Chris Wellons explains how to build strings in several steps. This is achieved thanks to the creation of a temporary buffer that is passed to helper methods using the "current" buffer mechanism. A nice and simple design pattern for Emacs.

Libraries

[s.el: The long lost Emacs string manipulation library](#)

Tools

format\$ macro

The `format$` macro (currently hosted [here](#)) allows for easy string interpolation, including optional `%` sequences as used by `format`. For example, this:

```
(format$ "Amount: ${amount% .02f} $name %s" date)
```

Expands to:

```
(format "Amount: % .02f %s %s" amount name date)
```

Since this happens at macro expansion time rather than at runtime, there is no performance penalty, in contrast to using `s-lex-format`.

Testing

- [Articles](#)
 - [Continuous Integration of Emacs Packages with CircleCI](#)

- **Frameworks**
 - [buttercup: Behavior-Driven Emacs Lisp Testing](#)
 - [ecukes: Cucumber for Emacs](#)
 - [Emacs Lisp Regression Testing \(ERT\)](#)
- **Libraries**
 - [assess: Test support functions](#)
 - [ert-expectations](#)
 - [propcheck: Quickcheck/hypothesis style testing](#)
 - [with-simulated-input: Test interactive functions non-interactively](#)
 - [xtest: Extensions for ERT](#)
- **Tools**

Articles

Continuous Integration of Emacs Packages with CircleCI

:archive.is: <https://archive.vn/uidZr>

I was very inspired by Damien Cassou's great presentation during EmacsConf 2019 to write this post and I encourage you to check it out if you haven't already. In short, when writing packages for Emacs, it is best practice to run several quality tools on them, like syntax and documentation checkers, or even ERT Tests. But once these packages are public and pull requests start coming in, it is a huge time saver to have these same tools ran automatically and provide feedback to contributors. That's right, we're talking about Continuous Integration for Emacs packages.

Frameworks

Frameworks for writing, organizing, and running tests.

[buttercup: Behavior-Driven Emacs Lisp Testing](#)

Buttercup is a behavior-driven development framework for testing Emacs Lisp code. It allows to group related tests so they can share common set-up and tear-down code, and allows the programmer to "spy" on functions to ensure they are called with the right arguments during testing.

The framework is heavily inspired by Jasmine.

[ecukes: Cucumber for Emacs](#)

There are plenty of unit/regression testing tools for Emacs, and even some for functional testing. What Emacs is missing though is a really good testing framework for integration testing. This is where Ecukes comes in.

Cucumber is a great integration testing tool, used mostly for testing web applications. Ecukes is Cucumber for Emacs. No, it's not a major mode to edit feature files. It is a package that makes it possible to write Cucumber like tests for your Emacs packages.

[Emacs Lisp Regression Testing \(ERT\)](#)

This is the standard, built-in Emacs testing library, used by core code and third-party packages alike.

Libraries

Libraries that help with writing tests.

[assess: Test support functions](#)

Assess provides additional support for testing Emacs packages.

It provides:

- a set of predicates for comparing strings, buffers and file contents.
- explainer functions for all predicates giving useful output
- macros for creating many temporary buffers at once, and for restoring the buffer list.
- methods for testing indentation, by comparison or "roundtripping".
- methods for testing fontification.

Assess aims to be as stateless as possible, leaving Emacs unchanged whether the tests succeed or fail, with respect to buffers, open files and so on; this helps to keep tests independent from each other.

[ert-expectations](#)

`expectations` allows more concise definitions of ERT tests. For example:

```
;; With ERT:

(ert-deftest erte-test-00001 ()
  (should (equal 10 (+ 4 6))))

;; With Expectations:
(expect 10 (+ 4 6))

;; Or:
(expectations
 (desc "success")
 (expect 10 (+ 4 6))
 (expect 5 (length "abcde")))
(desc "fail")
(expect 11 (+ 4 6))
(expect 6 (length "abcde")))
```

propcheck: Quickcheck/hypothesis style testing

propcheck brings property based testing to Emacs. It's similar to the excellent Hypothesis library for Python.

with-simulated-input: Test interactive functions non-interactively

This package provides an Emacs Lisp macro, `with-simulated-input`, which evaluates one or more forms while simulating a sequence of input events for those forms to read. The result is the same as if you had evaluated the forms and then manually typed in the same input. This macro is useful for non-interactive testing of normally interactive commands and functions, such as `completing-read`.

Some interactive functions rely on idle timers to do their work, so you might need a way to simulate idleness. For that, there is the `wsim-simulate-idle-time` function. You can insert calls to this function in between input strings.

xtest: Extensions for ERT

XTest is a simple set of extensions for ERT. XTest speeds up the creation of tests that follow the "one assertion per test" rule of thumb. It also simplifies testing functions that manipulate buffers. XTest aims to do a few things well, instead of being a monolithic library that attempts to solve every conceivable testing need. XTest is designed to be paired with vanilla ERT and other ERT libraries, where the user mixes and matches depending on their needs.

Tools

- See [Building / Testing tools](#).

User interface

- [Libraries](#)
 - [bui](#): Buffer interface library
 - [lister](#): Yet another list printer
 - [calfw](#): Calendar framework
 - [ctable](#): Table Component
 - Emacs's Widget for Object Collections (ewoc)
 - [hydra](#)
 - [navigel](#)
 - [tabulated-list-mode](#)
 - [Transient](#)
 - [tui](#): An experimental text-based UI framework modeled after React
 - [widget](#)
 - [widget-mvc](#): Web-like MVC framework

Libraries

bui: Buffer interface library

BUI (Buffer User Interface) is an Emacs library that can be used to make user interfaces to display some kind of entries (like packages, buffers, functions, etc.).

The intention of BUI is to be a high-level library which is convenient to be used both by:

- package makers, as there is no need to bother about implementing routine details and usual features (like buffer history, filtering displayed entries, etc.);
- users, as it provides familiar and intuitive interfaces with usual keys (for moving by lines, marking, sorting, switching between buttons); and what is also important, the defined interfaces are highly configurable through various generated variables. A summary of available key bindings can be displayed by pressing `h`.

Usage

BUI provides means to display entries in 2 types of buffers:

- `list`: it is based on `tabulated-list-mode`, thus it looks similar to a list of Emacs packages (`M-x list-packages`);
- `info`: it can be used to display more verbose info, like various buttons, text and other stuff related to the displayed entry (or entries).

In short, you define how a `list` / `info` interface looks like (using `bui-define-interface` macro), and then you can make some user commands that will display entries (using `bui-get-display-entries` and similar functions).

lister: Yet another list printer

Lister is a library for creating interactive "lists" of any kind. In contrast to similar packages like `hierarchy.el` or `tablist.el`, it aims at not simply mapping a data structure to a navigatable list. Rather, it treats the list like Emacs treats buffers: It is an empty space to which you can successively add stuff. So in Emacs lingo, `lister` should be rather called `listedit` - it is a library for editing lists, instead of displaying them.

calfw: Calendar framework

This program displays a calendar view in the Emacs buffer.

It is also usable as a library to display items on a calendar.

ctable: Table Component

ctable.el is a table component for Emacs Lisp. Emacs Lisp programs can display a nice table view from an abstract data model. The many emacs programs have the code for displaying table views, such as dired, list-process, buffer-list and so on. So, ctable.el would provide functions and a table framework for the table views.

Emacs's Widget for Object Collections (ewoc)

The Ewoc package constructs buffer text that represents a structure of Lisp objects, and updates the text to follow changes in that structure. This is like the “view” component in the “model–view–controller” design paradigm. Ewoc means “Emacs’s Widget for Object Collections”.

An ewoc is a structure that organizes information required to construct buffer text that represents certain Lisp data. The buffer text of the ewoc has three parts, in order: first, fixed header text; next, textual descriptions of a series of data elements (Lisp objects that you specify); and last, fixed footer text.

- [Manual](#)

hydra

This is a package for GNU Emacs that can be used to tie related commands into a family of short bindings with a common prefix - a Hydra.

navigel

The navigel package is a library that makes it simpler for Emacs Lisp developers to define user-interfaces based on tablists (also known as tabulated-lists). Overriding a few (CL) methods and calling `navigel-open` is all that's required to get a nice UI to navigate your domain objects (files, music library, database, etc.).

tabulated-list-mode

Tabulated List mode is a major mode for displaying tabulated data, i.e., data consisting of entries, each entry occupying one row of text with its contents divided into columns. Tabulated List mode provides facilities for pretty-printing rows and columns, and sorting the rows according to the values in each column.

- [Manual](#)

Transient

The library that powers Magit's command/option UI.

Taking inspiration from prefix keys and prefix arguments, Transient implements a similar abstraction involving a prefix command, infix arguments and suffix commands.

tui: An experimental text-based UI framework modeled after React

This is an experiment in building purely text-based user interfaces (TUI's). The ultimate goal is to explore new paradigms for user interface design and development using Emacs. To this end, tui.el implements an API based on the popular React JavaScript framework in order to reduce the demands involved with designing and building complex text-based UI's. This is all currently experimental! Expect things to change as I get feedback about what works, what does not!

widget

Most graphical user interface toolkits provide a number of standard user interface controls (sometimes known as “widgets” or “gadgets”). Emacs doesn't really support anything like this, except for an incredibly powerful text “widget.” On the other hand, Emacs does provide the necessary primitives to implement many other widgets within a text buffer. The `widget` package simplifies this task.

- [Manual](#)

widget-mvc: Web-like MVC framework

This is a GUI framework for Emacs Lisp. It is designed for programmers who are familiar with conventional Web MVC frameworks.

Version control

Tools

Magit

One of the “killer apps” for Emacs—and for git!

XML / HTML

Libraries

- [esxml](#)
- [elquery](#)
- [elfeed/xml-query.el](#)
- [enlive](#)
- [xml-plus](#)
- [xmlgen: An s-expression to XML DSL](#)

These libraries can all be used for HTML.

esxml

Probably the most featureful, usable library at the moment.

This library provides to formats for xml code generation. The primary form is esxml. esxml is the form that is returned by such functions as libxml-parse-xml-region and is used internally by emacs in many xml related libraries.

It also provides esxml-query :

```
;; Traditionally people pick one of the following options when faced
;; with the task of extracting data from XML in Emacs Lisp:
;;
;; - Using regular expressions on the unparsed document
;; - Manual tree traversal with `assoc', `car' and `cdr'
;;
;; Browsers faced a similar problem until jQuery happened, shortly
;; afterwards they started providing the `node.querySelector' and
;; `node.querySelectorAll' API for retrieving one or all nodes
;; matching a given CSS selector. This code implements the same API
;; with the `esxml-query' and `esxml-query-all' functions. The
;; following table summarizes the currently supported modifiers and
;; combinators:
;;
;; | Name | Supported? | Syntax |
;; |-----+-----+-----+
;; | Namespaces | No | foo|bar |
;; | Commas | Yes | foo, bar |
;; | Descendant combinator | Yes | foo bar |
;; | Child combinator | Yes | foo>bar |
;; | Adjacent sibling combinator | No | foo+bar |
;; | General sibling combinator | No | foo~bar |
;; | Universal selector | Yes | * |
;; | Type selector | Yes | tag |
;; | ID selector | Yes | #foo |
;; | Class selector | Yes | .foo |
;; | Attribute selector | Yes | [foo] |
;; | Exact match attribute selector | Yes | [foo=bar] |
;; | Prefix match attribute selector | Yes | [foo^=bar] |
;; | Suffix match attribute selector | Yes | [foo$=bar] |
;; | Substring match attribute selector | Yes | [foo*=bar] |
;; | Include match attribute selector | Yes | [foo~=bar] |
;; | Dash match attribute selector | Yes | [foo|=bar] |
;; | Attribute selector modifiers | No | [foo=bar i] |
;; | Pseudo elements | No | ::foo |
;; | Pseudo classes | No | :foo |
```

Example:

```
(defun org-books--amazon (url)
  "Return plist of data for book at Amazon URL."
  (cl-flet ((field (target-field list)
    (cl-loop for li in list
      for (field value) = (ignore-errors
        (-let (((_ _ field) value) li))
        (list field value)))
      when (equal field target-field)
      return (s-trim value))))
    (let* ((html (org-web-tools--get-url url))
      (tree (with-temp-buffer
        (insert html)
        (libxml-parse-html-region (point-min) (point-max)))))
      (author (esxml-query "span.author a.contributorNameID *" tree))
      (title (esxml-query "div#booksTitle h1#title > span *" tree))
      (details (esxml-query-all "table#productDetailsTable ul li" tree))
      (date (if-let ((printed (third (esxml-query-all "div#booksTitle h1#title span *" tree))))
        ;; Printed book
        (s-replace "- " "" printed)
        ;; Kindle book
        (field "Publication Date:" details)))
      (asin (field "ASIN:" details))
      (publisher (-some->> (field "Publisher:" details)
        (replace-regexp-in-string (rx " (" (1+ anything) ")") "")))
      (isbn-10 (field "ISBN-10:" details))
      (isbn-13 (field "ISBN-13:" details)))
      (list :author author :title title :publisher publisher :date date
        :asin asin :isbn-10 isbn-10 :isbn-13 isbn-13))))
```

elquery

It's like jQuery, but way less useful.

Example:

```
<html style="height: 100vh">
<head class="kek"><title class="kek" data-bar="foo">Complex HTML Page</title></head>
<body class="kek bur" style="height: 100%">
<h1 id="bar" class="kek wow">Wow this is an example</h1>
<input id="quux" class="kek foo"/>
<iframe id="baz" sandbox="allow-same-origin allow-scripts allow-popups allow-forms"
  width="100%" height="100%" src="example.org">
</iframe>
</body>
</html>
```

```
(let ((html (elq-read-file "~/kek.html")))
  (elq-el (car (elq-$ ".kek#quux" html))) ; => "input"
  (mapcar 'elq-el (elq-$ ".kek" html)) ; => ("input" "h1" "body" "title" "head")
  (mapcar (lambda (el) (elq-el (elq-parent el)))
    (elq-$ ".kek" html)) ; => ("body" "body" "html" "head" "html")
  (mapcar (lambda (el) (mapcar 'elq-el (elq-siblings el)))
    (elq-$ ".kek" html)) ; => (("h1" "input" "iframe") ("h1" "input" "iframe") ("head" "body")
  (elq-$ ".kek" html) ; => Hope you didn't like your messages buffer
  (elq-write html nil)) ; => "<html style=\"height: 100vh\"> ... </html>"
```

elfeed/xml-query.el

Provides lisp-based (rather than string-based) selectors. This library is primarily aimed at internal `elfeed` use rather than general use, however it may be useful to others. The author is [considering](#) publishing it separately.

```
;; Grab the top-level paragraph content from XHTML.
(xml-query-all '(html body p *) xhtml)

;; Extract all the links from an Atom feed.
(xml-query-all '(feed entry link [rel "alternate"] :href) xml)
```

enlive

This provides a limited set of lisp-based selectors (rather than string-based selectors).

Example:

```
(require 'enlive)

(enlive-text
  (enlive-query (enlive-fetch "http://gnu.org/") [title])) ; => "The GNU Operating System and the Fre
```

xml-plus

Mostly undocumented, providing three main functions:

```
;; Utility functions for xml parse trees.
;; - `xml+-query-all' and `xml+-query-first' are query functions that search
;; descendants in node lists. They don't work with namespace-aware parsing yet
;;
;; - `xml+-node-text' gets node text
```

xmlgen: An s-expression to XML DSL

Generate XML using sexps with the function `xmlgen` :

```
(xmlgen '(html
  (head
    (title "hello")
    (meta :something "hi"))
  (body
    (h1 "woohhooo")
    (p "text")
    (p "more text"))))
```

produces this:

```
<html>
<head>
  <title>hello</title>
  <meta something="hi" />
</head>
<body>
  <h1>woohhooo</h1>
  <p>text</p>
  <p>more text</p>
</body>
</html>
```

Blogs

Planet Emacsen

This is the main community aggregator. You can find just about everyone's Emacs-related blog posts here.

Sacha Chua's /Emacs News/

This is Sacha's weekly Emacs news digest. Don't miss it!

Artur Malabarba's /Endless Parentheses/

Irreal

One of the top Emacs blogs, frequently updated, and often highlights other interesting blog entries in the community.

Oleh Krehel's [/\(or emacs/](#)

Sacha Chua's [/Living an Awesome Life/](#)

People

The Emacs community is so full of brilliant, generous people that I can't keep track of them all! I will surely overlook many, and I will add them in no particular order, but merely as I come across them again and again.

- [Anders Lindgren](#)
- [Artur Malabarba](#)
- [Chris Wellons](#)
- [Damien Cassou](#)
- [Henrik Lissner](#)
- [John Wiegley](#)
- [Jonas Bernoulli](#)
- [Jorgen Schäfer](#)
- [Magnar Sveen](#)
- [Matus Goljer](#)
- [Oleh Krehel](#)
- [Phil Lord](#)
- [Roland Walker](#)
- [Sacha Chua](#)
- [Wilfred Hughes](#)

Anders Lindgren

Anders, aka Lindydancer, has written numerous packages to help with developing highlighting and font-lock packages, as well as some other useful tools.

- [GitHub](#)

Packages

el2markdown: Convert Emacs Lisp comments to Markdown

face-explorer: Library and tools for faces and text properties

faceup: Regression test system for font-lock keywords

font-lock-profiler: Coverage and timing tool for font-lock keywords

font-lock-regression-suite: Regression test suite for font-lock keywords of Emacs standard modes

font-lock-studio: Debugger for Font Lock keywords

highlight-refontification: Visualize how font-lock refontifies a buffer

lisp-extra-font-lock: Highlight bound variables and quoted expressions in lisp

multicolumn: Multiple side-by-side windows support

Artur Malabarba

Another prolific Emacs contributor, package developer, and blogger.

- [Blog: /Endless Parentheses/](#)
- [GitHub](#)

Packages

aggressive-indent-mode

paradox

Chris Wellons

Chris is the author of packages like [Elfeed](#), [EmacSQL](#), and [aio](#). He also writes about Emacs at his blog, [null program](#).

- [Blog: /null program/](#)
- [GitHub](#)

Damien Cassou

- [GitHub](#)

Packages

`beginend.el`

`navigel`

Henrik Lissner

- State “TODO” from [2020-01-03 Fri 06:27]

Author and maintainer of Doom Emacs, one of the most popular configurations.

- [GitHub](#)

Packages

[doom-emacs](#): An Emacs configuration for the stubborn martian vimmer

[emacs-doom-themes](#): An opinionated pack of modern color-themes

[emacs-doom-themer](#)

John Wiegley

John is the current Emacs maintainer.

- [Blog](#)
- [GitHub](#)

Packages

`use-package`

Jonas Bernoulli

Jonas is a prolific Emacs package developer and maintainer. You could spend hours on his GitHub repo.

- [GitHub](#)

Packages

`Magit`

`ox-texinfo+`

Jorgen Schäfer

Packages

`buttercup`: Behavior-Driven Emacs Lisp Testing

[Circe](#), a Client for IRC in Emacs

[elpy](#): Emacs Python Development Environment

[pyvenv](#): Python virtual environment interface

Magnar Sveen

- [GitHub](#)

Packages

`dash.el`

`expand-region.el`

`multiple-cursors.el`

`s.el`

Matus Goljer

- [GitHub](#)

Packages

`dash.el`

`smartparens`

Oleh Krehel

Oleh is a prolific package author, having contributed many very high-quality packages. He also writes at his blog.

Packages

ace-window: Quickly switch windows

avy: Jump to things tree-style

lispy: short and sweet LISP editing

swiper: Ivy - a generic completion frontend, Swiper - isearch with an overview, and more. Oh, man!

Phil Lord

- [GitHub](#)

Packages

lentic: Create views of the same content in two buffers

m-buffer-el

Roland Walker

Roland has published a wide variety of useful Emacs packages.

- [GitHub](#)

Packages

list-utils: List-manipulation utility functions

Sacha Chua

Sacha could easily be nominated the official Emacs ambassador, were there to be one. Her contributions to the Emacs and Org-mode communities are innumerable. One of her greatest recent contributions is her weekly [Emacs news](#) posts that serve as a digest of everything that happened in the Emacs world over the past week.

- [Blog: /Living an Awesome Life/](#)
- [GitHub](#)

Wilfred Hughes

Wilfred has published several useful packages, and he's also leading the [Rust Emacs port](#).

Packages

emacs-refactor

ht.el

suggest.el

Contributions

Yes, please! Send pull requests and file issues on the [GitHub repo](#). This is intended to be a community project.

Guidelines

Catalog and tag appropriately

New entries in the outline should have the appropriate tags and should follow the existing hierarchy. For example, articles should be tagged `articles`, and generally filed under an `Articles` heading using tag inheritance to apply the tag.

“Loosely” or “usefully” opinionated

Rather than being a place to dump links for users to sort out, we should do that ourselves. Links should have summaries and examples. Where there are multiple links to similar projects, we should compare them and guide users to what we think is generally the best solution, or the best solution for each use case.

Archive reference material

Much of the shared wisdom in the Emacs community is written in a variety of blogs by users and developers, as well as posts on Reddit, StackOverflow, etc. These tend to hang around for a long time, but being the Internet, this is never guaranteed. When linking to an article or other reference material, we should store a link to an archived version using [this code](#).

Requirements

org-make-toc

This package updates the table of contents. It's automatically used by this document through file-local variables, which you should be prompted to allow when opening the file.

Tasks

These resources should be added to the appropriate sections above. Since it takes some work to catalog and organize them, they are dumped here for future reference. Pull requests for these are welcome!

Benchmark seq-filter against cl-loop, et al

See [Optimize by minad · Pull Request #30 · tarsius/minions · GitHub](#).

Add el-search

Useful for searching Elisp code, doing query/replace on it in a Lisp-aware way, etc.

stream.el benchmarks

[2020-11-03 Tue 15:27] It's very surprising that `stream` seems faster than a plain `cl-loop`. I wonder if I'm doing something wrong...

Also see:

- [Has anything of substance ever been done in elisp using lazy data structures? : emacs](#)

```
(bench-multi-lexical :times 100 :ensure-equal t
:forms ((("cl-loop"
  (let* ((buffer (find-file-noselect "~/org/main.org"))
        (regexp (rx bow "Emacs" eow)))
    (with-current-buffer buffer
      (goto-char (point-min))
      (length (cl-loop while (re-search-forward regexp nil t)
                        collect (match-string-no-properties 0)))))))

  ("stream-regexp/seq-do/push"
  (let* ((buffer (find-file-noselect "~/org/main.org"))
        (regexp (rx bow "Emacs" eow))
        (stream (stream-regexp buffer regexp))
        result)
    (with-current-buffer buffer
      (goto-char (point-min))
      (seq-do (lambda (_match)
                (push (match-string-no-properties 0) result))
              stream)
      (length result))))

  ("stream-regexp/cl-loop"
  (let* ((buffer (find-file-noselect "~/org/main.org"))
        (regexp (rx bow "Emacs" eow))
        (stream (stream-regexp buffer regexp)))
    (with-current-buffer buffer
      (goto-char (point-min))
      (length (cl-loop while (stream-pop stream)
                        collect (match-string-no-properties 0)))))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
stream-regexp/cl-loop	1.03	1.305895	1	0.193277
stream-regexp/seq-do/push	1.42	1.350986	1	0.199524
cl-loop	slowest	1.925070	0	0

[2020-11-03 Tue 16:57] Other stream-related tests (these methods are bespoke):

Buffer lines

The benchmark macros need to be extended to allow definitions that aren't part of the benchmarked code.

```
(cl-defmethod stream-lines ((buffer buffer) &optional pos no-properties)
  "Return a stream of the lines of the buffer BUFFER.
  BUFFER may be a buffer or a string (buffer name).
  The sequence starts at POS if non-nil, 'point-min' otherwise."
  ;; Copied from the buffer method.
  (let ((fn (if no-properties
                #'buffer-substring-no-properties
                #'buffer-substring)))
    (with-current-buffer buffer
      (unless pos (setq pos (point-min)))
      (if (>= pos (point-max))
          (stream-empty))
      (stream-cons
        (with-current-buffer buffer
          (save-excursion
            (save-restriction
              (widen)
              (goto-char pos)
              (progn1 (funcall fn (point-at-bol) (point-at-eol))
                (setf pos (progn
                           (forward-line 1)
                           (point)))))))
          (stream-lines buffer pos no-properties))))))

(bench-multi-lexical :times 100 :ensure-equal t
:forms ((("stream-lines/seq-take/seq-into"
  (let* ((buffer (find-file-noselect "~/org/main.org"))
        (stream (stream-lines buffer nil 'no-properties)))
    (seq-into (seq-take stream 10) 'list)))
```

```
("cl-loop"
  (let* ((buffer (find-file-noselect "~/org/main.org")))
    (no-properties t))
  (with-current-buffer buffer
    (save-excursion
      (save-restriction
        (widen)
        (goto-char (point-min))
        (cl-loop for fn = (if no-properties
                              #'buffer-substring-no-properties
                              #'buffer-substring)
                  collect (funcall fn (point-at-bol) (point-at-eol))
                  do (forward-line 1)
                  until (eobp))))))
```

String lines

The benchmark macros need to be extended to allow definitions that aren't part of the benchmarked code. I'm not sure how I even got those results because it's not working now...

```
(cl-defmethod stream-lines ((string string) &optional pos no-properties)
  "Return a stream of the lines of the string STRING.
The sequence starts at POS if non-nil, 0 otherwise."
  ;; Copied from the buffer method.
  (unless pos (setq pos 0))
  (let ((fn (if no-properties
                #'buffer-substring-no-properties
                #'buffer-substring))
        (eol (when (string-match "\n" string pos)
                  (match-beginning 0))))
    (stream-cons
      (seq-subseq string pos eol)
      (if (not eol)
          (stream-empty)
          (stream-lines string (1+ eol) no-properties)))))

(bench-multi-lexical :times 100 :ensure-equal t
  :forms ((("stream-lines/seq-into"
            (let* ((string "abcd
efgh
hijk
zz"))
              (seq-into (stream-lines string nil 'no-properties) 'list)))

            ("s-lines"
            (let* ((string "abcd
efgh
hijk
zz"))
              (s-lines string)))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
s-lines	10.47	0.000513	0	0
stream/seq-into	slowest	0.005370	0	0

Emacs Lisp Animations | Digital | Dan Torop

- State "TODO" from [2021-10-01 Fri 11:22]
- Emacs, a programmer's text editor with roots in the 1970s, is a great tool for animation. In Fall, 2010 I taught a digital art class at NYU's interdisciplinary Steinhardt art school. Clearly, the thing to do was to teach how to make animations in Emacs by programming it in Lisp.
- These exercises are meant for non-programmers to get a glimpse of what a program and a language can do, by creating "physical" objects, in this case punctuation marks on the screen. ASCII art is the future, yes? At the very least, ASCII art in the '70s was to computer science what post-minimalism was to the contemporary art of the period. Think of a pile of ampersands and exclamation marks as earth art.

AddGitHub - doublep/datetime: Library for parsing, formatting, matching and recoding timestamps and date-time format strings.

Not to be confused with the datetime-format library.

Mention Emacs 27.1 SVG screenshots

e.g. https://www.reddit.com/r/emacs/comments/idz35e/emacs_27_can_take_svg_screenshots_of_itself/

Add section about files

e.g. saving files to disk.

Best practices

Mention file-precious-flag

Discussions

Add Emacs manual's Appendix D "Tips and Conventions"

[GitHub - tarsius/map-regexp: Map over matches of a regular expression](#)

[GitHub - Wilfred/elisp-def: Find Emacs Lisp definitions](#)

- State "TODO" from [2021-09-06 Mon 04:06]

[GitHub - tarsius/map-progress: Mapping macros that report progress](#)

[GitHub - jasonm23/autothemer: Conveniently create Emacs themes](#)

[emacs/rtree.el at master · emacs-mirror/emacs · GitHub](#)

[GitHub - nicferrier/emacs-s-buffer: string operations on emacs buffers](#)

[GitHub - DarwinAwardWinner/emacs-named-timer: Simplified timer management for Emacs Lisp](#)

Add [GitHub - vermiculus/package-demo: Script your Emacs package demos!](#)

[GitHub - vermiculus/stash.el: Lightweight, Persistent Caching for Elisp](#)

`easy-mmode.el`

Especially `easy-mmode-defmap` .

[GitHub - rocky/elisp-decompile: Emacs Lisp Decompiler](#)

[GitHub - ellerh/peg.el](#)

Articles to add [0/13]

[Read and write files in Emacs Lisp](#) (5 min read)

[A Future For Concurrency In Emacs Lisp](#) (6 min read)

[A Blast From The Past: The Tale Of Concurrency In Emacs](#) (7 min read)

[I wished GNU Emacs had...](#) (2 min read)

[Reproduce bugs in emacs -Q](#) (4 min read)

[Why package.el?](#) (1 min read)

[My Emacs Configuration with use-package](#) (8 min read)

[Emacs script pitfalls](#) (13 min read)

[Autoloads in Emacs Lisp](#) (5 min read)

[Advanced syntactic fontification](#) (11 min read)

[Calling Python from Haskell](#) (12 min read)

[Search-based fontification with keywords](#) (18 min read)

[Syntactic fontification in Emacs](#) (10 min read)

Add people [1/8]

Add more of Roland Walker's packages

[Nic Ferrier](#)

[GitHub - nicferrier/emacs-db: very simple database for emacs-lisp, can also wrap other databases.](#)

[GitHub - nicferrier/emacs-noflet: noflet - nic's overriding flet, for fleting functions for the purpose of decorating them](#)

Sean Allred

Toby Cubitt

A variety of packages published, e.g. at http://www.dr-qubit.org/emacs_data-structures.html

Vasilij Schneidermann

Vincent Touns' projects

He has a lot of interesting libraries on his repo, and some of them are *extensively* documented. An aspiring Emacs Lisp developer could learn a lot from his code.

Clemens Radermacher

- [blog /With Emacs/](#)
- [GitHub](#)
 - [Packages](#)

Chris Wellons

- State "DONE" from "UNDERWAY" [2020-11-09 Mon 00:53]
- State "UNDERWAY" from "TODO" [2020-11-08 Sun 18:29]
- <http://github.com/skeeto>
- Elfeed
- emacs-ai: async/await for Emacs Lisp
- Other Emacs packages

Add tips for new developers

e.g.:

- Commonly used minor modes
 - `highlight-funcalls`
 - `highlight-quoted`
 - `outline-minor-mode`

elisp - A faster method to obtain `line-number-at-pos` in large buffers - Emacs Stack Exchange

Add <https://github.com/joddie/pcr2el>

Add [GitHub - bbatsov/emacs-lisp-style-guide](#): A community-driven Emacs Lisp style guide

Add MELPA

Mention [@milkypostman](#), [@purcell](#), [@syohex](#), etc. Mention [sandbox](#).

Add [GitHub - vermiculus/apiwrap.el](#): Generate wrappers for your API endpoints!

Add [Modern Emacs](#) site

Add [GitHub - sigma/pcache](#): persistent caching for Emacs

Dynamic modules section

[GitHub - jkitchin/emacs-modules](#): Dynamic modules for emacs

Add resources from its [readme](#)

For my own notes here are all the resources on dynamic modules I know of:

Here are the official Emacs header and example: `emacs-module.h`:
<http://git.savannah.gnu.org/cgiit/emacs.git/tree/src/emacs-module.h?id=e18ee60b02d08b2f075903005798d3d6064dc013> `mod_test.c`:
<http://git.savannah.gnu.org/cgiit/emacs.git/tree/modules/mod-test/mod-test.c?id=e18ee60b02d08b2f075903005798d3d6064dc013>

This simple example in C <http://diobla.info/blog-archive/modules-tut.html>

joymacs

<http://nullprogram.com/blog/2016/11/05/>

mruby

<https://github.com/syohex/emacs-mruby-test>

<https://github.com/tromey/emacs-ffi>

an actual ffi for emacs

elfuse

<https://github.com/vkazanov/elfuse> a file system in Emacs

asynchronous events

<http://nullprogram.com/blog/2017/02/14/> related to elfuse

emacs-sqlite3

sqlite3 binding of Emacs Lisp

emacs-parson

JSON parser with dynamic module feature with [parson](#)

libyaml

libyaml

emacs-perl

Embed Perl into Emacs

<https://github.com/syohex/emacs-eject>

eject a cd

emacs-capstone

elisp bindings for the [capstone](#) disassembler

emacs-csound

EmacsLisp link to Csound's API via Emacs Modules

emacs-cmigemo

Emacs dynamic module for cmigemo

emacs-cipher

OpenSSL cipher binding of Emacs Lisp

emacs-lua

Lua engine from Emacs Lisp

emacs-ztd

libzstd binding of Emacs Lisp

mem-cached

libmemcached

<https://coldnew.github.io/2d16cc25/>

in Chinese, but with code

A collection of module resources: <https://github.com/emacs-pe/emacs-modules>

- Nim <https://github.com/yuutayamada/nim-emacs-module>
- OCaml <https://github.com/janestreet/ecaml>
- Rust <https://github.com/lunaryorn/emacs-module.rs> https://github.com/jipe/emacs_module_bindings

golang

<https://github.com/sigma/go-emacs> writing modules in go

This may not be a dynamic module but claims an ffi haskell! <https://github.com/knupfer/haskell-emacs>

Documentation best practices

Describe things like exporting an Org readme to an Info manual, e.g. like Magit, `org-super-agenda`, etc.

Add databases section

[GitHub - skeeto/emacsqli: A high-level Emacs Lisp RDBMS front-end](#)

[GitHub - pekingduck/emacs-sqlite3-api: Native SQLite3 API for GNU Emacs](#)

[GitHub - syohex/emacs-sqlite3: sqlite3 binding of Emacs Lisp](#)

[GitHub - kiwanami/emacs-edbi: Database Interface for Emacs Lisp](#)

Add [GitHub - ijp/mbe.el: macros by example in elisp](#)

Tree-traversal

[GitHub - volrath/treepy.el: Generic tree traversing tools for Emacs Lisp](#)

- State "TODO" from [2017-09-06 Wed 00:21]

Test in MELPA sandbox

- State "TODO" from [2017-12-16 Sat 20:16]

[2017-07-29 Sat 00:33] Not only should you test installing and using your package in the sandbox, but you should *also* test then exiting the sandbox Emacs, running it again with the package already installed, and loading it. This is because, when the sandbox installs the package, the byte-compilation seems to load some things that won't be loaded the same way when only loading the byte-compiled file (especially if you have any `eval-when-compile` lines, or unusual macros or things that modify the environment when loaded).

Sequence shuffling examples and benchmarks

Benchmarking sequence shuffling

See <https://github.com/melpa/melpa/pull/6191#issuecomment-498101336>

```
(defun key-quiz--shuffle-list (list)
  "Shuffles LIST randomly, modifying it in-place."
  (dolist (i (reverse (number-sequence 1 (1- (length list)))))
    (let ((j (random (1+ i)))
          (tmp (elt list i)))
      (setf (elt list i) (elt list j))
      (setf (elt list j) tmp)))
  list)

(defun key-quiz--shuffle-list-nreverse (list)
  "Shuffles LIST randomly, modifying it in-place."
  (dolist (i (nreverse (number-sequence 1 (1- (length list)))))
    (let ((j (random (1+ i)))
          (tmp (elt list i)))
      (setf (elt list i) (elt list j))
      (setf (elt list j) tmp)))
  list)

(defun elfeed--shuffle (seq)
  "Destructively shuffle SEQ."
  (let ((n (length seq)))
    (prog1 seq
      (dotimes (i n)
        (cl-rotatef (elt seq i) (elt seq (+ i (random (- n i))))))))

(defun faster-seq-sort-by (function pred sequence)
  "Sort SEQUENCE using PRED as a comparison function.
Elements of SEQUENCE are transformed by FUNCTION before being
sorted. FUNCTION must be a function of one argument."
  ;; This version is modified to avoid calling "random" twice every time the predicate is called.
  (seq-map 'cdr
    (sort (seq-map (lambda (x) (cons (funcall function x) x)) sequence)
      (lambda (a b)
        (funcall pred (car a) (car b)))))

(defun seq-sort-by--shuffle (seq)
  (seq-sort-by (lambda (_) (random)) #'<= seq))

(defun faster-seq-sort-by--shuffle (seq)
  (faster-seq-sort-by (lambda (_) (random)) #'<= seq))
```

Lists

```
(let ((big-list (seq-into (seq-take obarray 5000) 'list)))
  (bench-multi-lexical :times 100
    :forms (("key-quiz--shuffle-list" (key-quiz--shuffle-list big-list))
            ("key-quiz--shuffle-list-nreverse" (key-quiz--shuffle-list-nreverse big-list))
            ("elfeed--shuffle" (elfeed--shuffle big-list))
            ("seq-sort-by--shuffle" (seq-sort-by--shuffle big-list))
            ("faster-seq-sort-by--shuffle" (faster-seq-sort-by--shuffle big-list)))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
faster-seq-sort-by--shuffle	1.38	1.725037	0	0
seq-sort-by--shuffle	15.01	2.378234	0	0
key-quiz--shuffle-list-nreverse	1.03	35.703316	27	17.892723
key-quiz--shuffle-list	1.24	36.630320	28	18.768216
elfeed--shuffle	slowest	45.439405	32	21.130538

Vectors

```
(let ((big-list (seq-into (seq-take obarray 5000) 'vector)))
  (bench-multi-lexical :times 100
    :forms (("key-quiz--shuffle-list" (key-quiz--shuffle-list big-list))
            ("key-quiz--shuffle-list-nreverse" (key-quiz--shuffle-list-nreverse big-list))
            ("elfeed--shuffle" (elfeed--shuffle big-list))
            ("seq-sort-by--shuffle" (seq-sort-by--shuffle big-list))
            ("faster-seq-sort-by--shuffle" (faster-seq-sort-by--shuffle big-list)))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
faster-seq-sort-by--shuffle	1.39	1.718990	0	0
seq-sort-by--shuffle	10.42	2.390860	0	0
key-quiz--shuffle-list-nreverse	1.02	24.918774	27	17.971779

key-quiz-shuffle-list	1.10	25.452665	28	18.487015
elfeed-shuffle	slowest	27.991305	32	21.215224

[GitHub - chrisbarrett/elisp-namespaces: UNMAINTAINED](#)

≡ README.org

- State "TODO" from [2019-10-05 Sat 12:20]

MAYBE Benchmarking seq-let vs pcase-let* with backquoted patterns

```
(bench-multi-lexical :times 1000 :ensure-equal t
  :forms ((("pcase-let* backquoted pattern"
    (pcase-let* ((` (,name ,argument) (list 'NAME 'ARGUMENT)))
      (list name argument)))

    ("seq-let"
      (seq-let (name argument) (list 'NAME 'ARGUMENT)
        (list name argument))))))
```

Form	x faster than next	Total runtime	# of GCs	Total GC runtime
pcase-let* backquoted pattern	2.73	0.000188	0	0
seq-let	slowest	0.000515	0	0

MAYBE [GitHub - raxod502/elint: 🐼 DEPRECATED: Small module to deduplicate Elisp build tooling.](#)

- State "MAYBE" from [2020-01-19 Sun 15:48]

He archived the project, so it probably shouldn't be used, but it may have interesting historical significance for developing similar tools.

Testing

Everything at [EmacsWiki: Unit Testing](#)

[GitHub - rejeep/el-mock.el: Mocking library for Emacs](#)

[GitHub - sigma/mocker.el: a simple mocking framework for Emacs](#)

[EmacsWiki: Emacs Lisp Expectations](#)

[How do you debug giant plists? : emacs](#)