Emacs Initialization Code

John Jacobsen

October 26, 2014

Contents

1 Introduction

Here's my Emacs init code. Some of it was adapted from Matthew Wampler-Doty. Other things have been borrowed from Sacha Chua and other places around the Internetz.

You'll find this code highly focused on Clojure and Org Mode. I have a lot of unconventional keybindings since I didn't bother to learn all the classic ones when I started with Emacs (probably sometime in the 1990s).

2 Utilities

Matt wrote a Clojure-like `filter` function. Probably I'll write more like it as I need them, since I am more used to Clojure than Emacs Lisp. (Why else would one bother with a "programmable programming language?"[1])

```
(defun filter (pred lst)
  "Use PRED to filter a list LST of elements."
  (delq nil (mapcar (lambda (x) (and (funcall pred x) x)) lst)))
```

3 Package management

MELPA is a bit too bleeding edge; since CIDER has broken for me before after upgrading, I use the more stable releases from `melpa-stable` only. To get back MELPA(-bleeding-edge), swap in the following:

```
;; (add-to-list 'package-archives
;;              '("MELPA" . "http://melpa.milkbox.net/packages/") t)
```

Define which remote archives to pull from, and which packages to use:

```
(require 'package)
(add-to-list
 'package-archives
 '("melpa-stable" . "http://melpa-stable.milkbox.net/packages/") t)
(add-to-list 'package-archives
             '("marmalade" . "http://marmalade-repo.org/packages/") t)
(package-initialize)
(defvar my-packages)
(setq my-packages
      '(company auto-complete autopair ac-cider cider color-theme
        zenburn-theme diminish goto-last-change main-line maxframe
        clojure-mode epl popup rainbow-delimiters smex
        undo-tree flycheck flycheck-hdevtools
```

[1] John Foderaro, CACM, Sept. 1991.

```
        git-timemachine paredit auto-indent-mode slamhound
        lorem-ipsum midje-mode hungry-delete metar))
```

Install any missing packages:

```
(let ((uninstalled-packages (filter (lambda (x) (not (package-installed-p x)))
                                    my-packages)))
  (when (and (not (equal uninstalled-packages '()))
             (y-or-n-p (format "Install packages %s?"  uninstalled-packages)))
    (package-refresh-contents)
    (mapc 'package-install uninstalled-packages)))
```

4 Startup

Some configuration to make things quieter on start up:

```
(setq inhibit-splash-screen t
      initial-scratch-message nil)
```

5 Initial keybindings

At some point I took it into my head to map most of my personal key bindings to combinations starting with `\C-o`. We have to set it free from its usual use (`open-line`) before we can use it as a prefix.

```
(global-unset-key "\C-o")
```

6 Remembering Where We Were Last Time Around

```
(require 'saveplace)
(setq-default save-place t)
(setq save-place-file "~/.emacs.d/saved-places")
```

7 General Lisp stuff

Rainbow delimiters for all programming major modes:

```
(require 'rainbow-delimiters)
(add-hook 'prog-mode-hook 'rainbow-delimiters-mode)
```

Show paren balancing nicely:

```
(require 'paren)
(set-face-background 'show-paren-match "white")
(add-hook 'prog-mode-hook 'show-paren-mode)
```

8 Stuff for Editing Emacs Lisp

I add a hook for evaluating the expression just before point; I've played with `auto-indent-mode` and `flycheck-mode` but tired of them. I do want `paredit` though (and therefore don't want `autopair-mode`).

```
(define-key emacs-lisp-mode-map (kbd "<s-return>") 'eval-last-sexp)

;;(add-hook 'emacs-lisp-mode-hook 'flycheck-mode)
;;(add-hook 'emacs-lisp-mode-hook 'auto-indent-mode)
(add-hook 'emacs-lisp-mode-hook
          (lambda ()
            (paredit-mode 1)
            (autopair-mode 0)))
```

9 Ruby stuff

Do two space indents for Ruby code:

```
(setq ruby-indent-level 2)
```

10 Backups

Tell Emacs to write backup files to their own directory, and make backups even for files in revision control:

```
(setq backup-directory-alist
      `(("." . ,(expand-file-name
                 (concat user-emacs-directory "backups")))))

(setq vc-make-backup-files t)
```

11 Autocomplete Mode

```
;; Autocomplete mode
(require 'auto-complete)
(add-hook 'prog-mode-hook 'auto-complete-mode)
```

12 λ

Make `lambda` show as λ:

```
(font-lock-add-keywords
 'emacs-lisp-mode
 '(("(\\(lambda\\)\\>"
    (0 (prog1 ()
         (compose-region (match-beginning 1)
                         (match-end 1)
                         ?))))))
```

13 Highlighting of long lines

```
(defun highlight-long-lines ()
  "Turn on highlighting of long lines."
  (interactive)
  (highlight-lines-matching-regexp ".\\{81\\}" 'hi-pink))


(defun unhighlight-long-lines ()
  "Turn off highlighting of long lines."
  (interactive)
  (unhighlight-regexp "^.*\\(?:.\\{81\\}\\).*$"))

;(global-set-key "\C-oH" 'highlight-long-lines)
;(global-set-key "\C-oh" 'unhighlight-long-lines)
```

14 Lots of keybindings

Many of these are extremely old, having followed me from machine to machine over the years. Some could probably be deleted.

```
(global-set-key [S-deletechar]  'kill-ring-save)
;; Set up the keyboard so the delete key on both the regular keyboard
;; and the keypad delete the character under the cursor and to the right
;; under X, instead of the default, backspace behavior.
(global-set-key [delete] 'delete-char)
(global-set-key [kp-delete] 'delete-char)

(define-key function-key-map "\e[1~" [find])
(define-key function-key-map "\e[2~" [insertchar])
(define-key function-key-map "\e[3~" [deletechar])
(define-key function-key-map "\e[4~" [select])
(define-key function-key-map "\e[5~" [prior])
(define-key function-key-map "\e[6~" [next])
(define-key global-map [select] 'set-mark-command)
(define-key global-map [insertchar] 'yank)
(define-key global-map [deletechar] 'kill-region)

(global-unset-key "\C- ")
(global-set-key [?\C- ] 'other-window)
(global-set-key "\C-A" 'split-window-horizontally)
(global-set-key "\C-oa" 'split-window-vertically)
(global-set-key "\C-K" 'kill-line)
(global-set-key "\C-os" 'isearch-forward-regexp)
(global-set-key "\C-oD" 'find-name-dired)
(global-set-key "\C-xS" 'sort-lines)
(global-set-key "\C-w" 'backward-kill-word)
(global-set-key "\C-x\C-k" 'kill-region)
(global-set-key "\C-c\C-k" 'kill-region)
(global-set-key "\C-ok" 'comment-region)
```

```
(global-set-key "\C-ou" 'uncomment-region)
(global-set-key "\C-oe" 'eval-current-buffer)
(global-set-key "\C-od" 'delete-horizontal-space)
(global-set-key "\C-ob" 'backward-word)
(global-set-key "\C-oq" 'query-replace-regexp)
(global-set-key "\C-oL" 'lorem-ipsum-insert-paragraphs)
(global-set-key "\C-]"  'fill-region)
(global-set-key "\C-ot" 'beginning-of-buffer)
(global-set-key "\C-oT" 'toggle-window-split)
(global-set-key "\C-N" 'enlarge-window)
(global-set-key "\C-o\C-n" 'enlarge-window-horizontally)
(global-set-key "\C-oc" 'paredit-duplicate-closest-sexp)
(global-set-key "\C-ol" 'goto-line)
(global-set-key "\C-ob" 'end-of-buffer)
(global-set-key "\C-op" 'fill-region)
(global-set-key "\C-og" 'save-buffers-kill-emacs)
(global-set-key "\C-od" 'downcase-region)
(global-set-key "\C-oR" 'indent-region)
(global-set-key "\C-or" 'rgrep)
(global-set-key "\C-L" 'delete-other-windows)
(global-set-key "\C-B" 'scroll-down)
(global-set-key "\C-F" 'scroll-up)
(global-set-key "\C-V" 'save-buffer)
(global-set-key "\C-R" 'isearch-forward)
(global-set-key "\C-^" 'wnt-alog-add-entry)
(global-set-key "\C-T" 'set-mark-command)
(global-set-key "\C-Y" 'yank)
(global-set-key "\C-D" 'backward-delete-char-untabify)
(global-set-key "\C-\\" 'shell)
(global-set-key "\C-oi" 'quoted-insert)
(global-set-key "\e[1~" 'isearch-forward)
(global-set-key [select] 'set-mark-command)
(global-set-key [insertchar] 'yank)
(global-set-key [deletechar] 'kill-region)
(global-set-key "\C-\\" 'shell)
(global-set-key "\C-oi" 'quoted-insert)
(global-set-key "\e[1~" 'isearch-forward)
(global-set-key [select] 'set-mark-command)
(global-set-key [insertchar] 'yank)
(global-set-key [deletechar] 'kill-region)
(global-set-key (kbd "s-0") 'org-todo-list)
```

Shortcuts for jumping directly into most commonly-used buffers:

```
(global-set-key "\C-oO" (lambda ()
                          (interactive)
                          (find-file "~/Dropbox/org/toplevel.org")))
(global-set-key "\C-oE" (lambda ()
                          (interactive)
                          (find-file "~/.emacs.d/org/init.org")))
```

Keyboard shortcuts for joining lines before and after point (thanks to http://whattheemacsd.com/ for the `(join-line -1) trick`):

```
(global-set-key (kbd "M-j")
  (lambda () (interactive) (join-line -1)))
(global-set-key "\C-oo" 'join-line)
```

Show trailing whitespace, 'cause *we hates it.*...

```
(setq-default show-trailing-whitespace t)
```

15 Clojure setup

Don't go to REPL buffer when starting Cider:

```
(setq cider-repl-pop-to-buffer-on-connect nil)
```

15.1 Key bindings special to Midje facts

Set up Midje fact with mark inserted at beginning of comment text (refill as needed in appropriate columns, using `C-oF`).

```
(global-set-key "\C-of" (lambda ()
                          (interactive)
                          (insert "(fact                                \"\"\n\n  )")
                          (backward-char 6)
                          (set-mark (point))))
```

Perform the refill operation for the text string in a Midje fact:

```
(global-set-key "\C-oF" (lambda ()
                          (interactive)
                          (set-left-margin (mark) (point) 37)
                          (fill-region (mark) (point))))
```

Append result of evaluating previous expression (Clojure):

```
(defun cider-eval-last-sexp-and-append ()
  "Evaluate the expression preceding point and append result."
  (interactive)
  (let ((last-sexp (cider-last-sexp)))
    ;; we have to be sure the evaluation won't result in an error
    (cider-eval-and-get-value last-sexp)
    (with-current-buffer (current-buffer)
      (insert ";;=>\n"))
    (cider-interactive-eval-print last-sexp)))


(defun cider-format-with-out-str-pprint-eval (form)
  "Return a string of Clojure code that will return pretty-printed FORM."
  (format "(clojure.core/let [x %s] (with-out-str (clojure.pprint/pprint x)))" form))
```

```
(defun cider-eval-last-sexp-and-pprint-append ()
  "Evaluate the expression preceding point and append pretty-printed result."
  (interactive)
  (let ((last-sexp (cider-last-sexp)))
    ;; we have to be sure the evaluation won't result in an error
    (with-current-buffer (current-buffer)
      (insert ";;=>\n")
      (insert (cider-eval-and-get-value (cider-format-with-out-str-pprint-eval last-sexp))))))


;; A few paredit things, also from whattheemacsd.com:
(defun paredit--is-at-start-of-sexp ()
  (and (looking-at "(\\|\\[")
       (not (nth 3 (syntax-ppss))) ;; inside string
       (not (nth 4 (syntax-ppss))))) ;; inside comment

(defun paredit-duplicate-closest-sexp ()
  (interactive)
  ;; skips to start of current sexp
  (while (not (paredit--is-at-start-of-sexp))
    (paredit-backward))
  (set-mark-command nil)
  ;; while we find sexps we move forward on the line
  (while (and (bounds-of-thing-at-point 'sexp)
              (<= (point) (car (bounds-of-thing-at-point 'sexp)))
              (not (= (point) (line-end-position))))
    (forward-sexp)
    (while (looking-at " ")
      (forward-char)))
  (kill-ring-save (mark) (point))
  ;; go to the next line and copy the sexprs we encountered
  (paredit-newline)
  (yank)
  (exchange-point-and-mark))
```

15.2 Correcting single-whitespaced toplevel forms

```
(defun correct-single-whitespace ()
  "Correct single-spaced Lisp toplevel forms."
  (interactive)
  (goto-char 1)
  (while (search-forward-regexp ")\n\n(" nil t)
    (replace-match ")\n\n\n(" t nil)))
(global-set-key "\C-oQ" 'correct-single-whitespace)

(add-hook 'clojure-mode-hook
          '(lambda ()
             (paredit-mode 1)
             (highlight-long-lines)
             (define-key clojure-mode-map (kbd "C-c e") 'shell-eval-last-expression)
```

```
            (define-key clojure-mode-map (kbd "C-o x") 'cider-eval-defun-at-point)
            (define-key clojure-mode-map (kbd "C-o j") 'cider-jack-in)
            (define-key clojure-mode-map (kbd "C-o J") 'cider-restart)
            (define-key clojure-mode-map (kbd "C-<up>") 'paredit-backward)
            (define-key clojure-mode-map (kbd "C-<down>") 'paredit-forward)
            (define-key clojure-mode-map (kbd "C-o y")
              'cider-eval-last-sexp-and-append)
            (define-key clojure-mode-map (kbd "C-o Y")
              'cider-eval-last-sexp-and-pprint-append)
            (define-key clojure-mode-map (kbd "s-i") 'cider-eval-last-sexp)
            (define-key clojure-mode-map (kbd "C-c x") 'shell-eval-defun)))

;; Minibuffer size
(add-hook 'minibuffer-setup-hook 'my-minibuffer-setup)
(defun my-minibuffer-setup ()
  (set (make-local-variable 'face-remapping-alist)
       '((default :height 1.5))))

;;;; Swap window split orientation
;;;; (http://emacs.stackexchange.com/questions/318/switch-window-split-orientation-fastest-way):
(defun toggle-window-split ()
  (interactive)
  (if (= (count-windows) 2)
      (let* ((this-win-buffer (window-buffer))
             (next-win-buffer (window-buffer (next-window)))
             (this-win-edges (window-edges (selected-window)))
             (next-win-edges (window-edges (next-window)))
             (this-win-2nd (not (and (<= (car this-win-edges)
                                         (car next-win-edges))
                                     (<= (cadr this-win-edges)
                                         (cadr next-win-edges)))))
             (splitter
              (if (= (car this-win-edges)
                     (car (window-edges (next-window))))
                  'split-window-horizontally
                'split-window-vertically)))
        (delete-other-windows)
        (let ((first-win (selected-window)))
          (funcall splitter)
          (if this-win-2nd (other-window 1))
          (set-window-buffer (selected-window) this-win-buffer)
          (set-window-buffer (next-window) next-win-buffer)
          (select-window first-win)
          (if this-win-2nd (other-window 1))))))
```

15.3 Mode line hack

Shorten `clojure-mode` in mode line[2].

```
(defmacro rename-modeline (package-name mode new-name)
```

[2]From http://whattheemacsd.com/

```
  `(eval-after-load ,package-name
     '(defadvice ,mode (after rename-modeline activate)
        (setq mode-name ,new-name))))

(rename-modeline "clojure-mode" clojure-mode "Clj")
```

16 Shell stuff, for running shells within Emacs

16.1 Path Magic

Smooth the waters for starting processes from the shell. "Set up Emacs' 'exec-path' and PATH environment variable to match the user's shell. This is particularly useful under Mac OSX, where GUI apps are not started from a shell[3]."

```
(defun set-exec-path-from-shell-PATH ()
  (interactive)
  (let ((path-from-shell
         (replace-regexp-in-string
          "[ \t\n]*$" ""
          (shell-command-to-string "$SHELL --login -i -c 'echo $PATH'"))))
    (setenv "PATH" path-from-shell)
    (setq exec-path (split-string path-from-shell path-separator))))
```

16.2 Moar Shells

Create shell in new buffer when needed, rather than just loading up the existing shell buffer.

```
(defun create-shell-in-new-buffer ()
  (interactive)
  (let ((currentbuf (get-buffer-window (current-buffer)))
        (newbuf (generate-new-buffer-name "*shell*")))
    (generate-new-buffer newbuf)
    (set-window-dedicated-p currentbuf nil)
    (set-window-buffer currentbuf newbuf)
    (shell newbuf)))

(global-set-key "\C-oS" 'create-shell-in-new-buffer)
```

16.3 Kill shell buffers quickly

"With this snippet, [a second] press of C-d will kill the buffer. It's pretty nice, since you then just tap C-d twice to get rid of the shell and go on about your merry way[4]"

```
(defun comint-delchar-or-eof-or-kill-buffer (arg)
  (interactive "p")
  (if (null (get-buffer-process (current-buffer)))
      (kill-buffer)
    (comint-delchar-or-maybe-eof arg)))
```

[3] See http://stackoverflow.com/questions/8606954/path-and-exec-path-set-but-emacs-does-not-find-executable

[4] From http://whattheemacsd.com.

```
(add-hook 'shell-mode-hook
          (lambda ()
            (define-key shell-mode-map
              (kbd "C-d") 'comint-delchar-or-eof-or-kill-buffer)))
```

17 Stuff related to configuring Emacs-in-a-window

When running GUI Emacs (i.e. on OS-X, which is the only way I run Emacs these days anyways), set the theme to Zenburn, turn off visual noise, fix up the PATH for shells, and allow resizing of window.

```
(when window-system
  (load-theme 'zenburn t)
  (tool-bar-mode -1)
  (scroll-bar-mode -1)
  (set-exec-path-from-shell-PATH)
  (global-set-key (kbd "s-=") 'text-scale-increase)
  (global-set-key (kbd "s--") 'text-scale-decrease))
```

18 Common Lisp

I haven't done too much Common Lisp programming yet, but have just played around. So far I find Emacs integration to be at least as good as with Clojure. Here I mimic two of the keybindings I use most from Clojure.

```
(require 'slime-autoloads)
(setq inferior-lisp-program "/usr/local/bin/sbcl")
(setq slime-contribs '(slime-fancy))
(add-hook 'lisp-mode-hook
          '(lambda ()
             (paredit-mode 1)
             (highlight-long-lines)
             (define-key lisp-mode-map (kbd "C-o j") 'slime)
             (define-key lisp-mode-map (kbd "s-i")
                         'slime-eval-last-expression)))
```

19 Org Mode

Put clock in/out timestamps into drawer, so they stay hidden when expanding items.

```
(setq org-clock-into-drawer t)
```

Set Clojure backend for literate programming.

```
(setq org-babel-clojure-backend 'cider)
```

Don't ask for confirmation before evaluating code in these languages (**use at your own risk**):

```
(defun my-org-confirm-babel-evaluate (lang body)
  (and
   (not (string= lang "lisp"))
   (not (string= lang "emacs-lisp"))
   (not (string= lang "clojure"))))
(setq org-confirm-babel-evaluate 'my-org-confirm-babel-evaluate)
```

Much general Org setup. . . :

```
(require 'org)
(require 'ob-clojure)

;; From http://sachachua.com/blog/2007/12/clocking-time-with-emacs-org/:
(eval-after-load 'org
  '(progn
     (defun wicked/org-clock-in-if-starting ()
       "Clock in when the task is marked STARTED."
       (when (and (string= org-state "STARTED")
                  (not (string= org-last-state org-state)))
         (org-clock-in)))
     (add-hook 'org-after-todo-state-change-hook
               'wicked/org-clock-in-if-starting)
     (defadvice org-clock-in (after wicked activate)
       "Set this task's status to 'STARTED'."
       (org-todo "STARTED"))
     (defun wicked/org-clock-out-if-waiting ()
       "Clock out when the task is marked WAITING."
       (when (and (string= org-state "WAITING")
                  (equal (marker-buffer org-clock-marker) (current-buffer))
                  (< (point) org-clock-marker)
                  (> (save-excursion (outline-next-heading) (point))
                     org-clock-marker)
                  (not (string= org-last-state org-state)))
         (org-clock-out)))
     (add-hook 'org-after-todo-state-change-hook
               'wicked/org-clock-out-if-waiting)))

(setq org-agenda-files '("~/Dropbox/org"))
(setq org-log-done t)
(setq org-refile-targets (quote ((nil :maxlevel . 10)
                                 (org-agenda-files :maxlevel . 10))))
(setq org-refile-use-outline-path t)
(setq org-outline-path-complete-in-steps nil)
(setq org-refile-allow-creating-parent-nodes (quote confirm))
(setq org-todo-keywords
      '((sequence "TODO" "STARTED" "WAITING" "SOMEDAY" "DONE")))
(define-key global-map "\C-ca" 'org-agenda)
```

'Remember' stuff[5]. TODO: use Org's own capture system.

```
(setq org-remember-templates
      '(("Tasks" ?t "* TODO %?\n  %i\n  %a" "~/Dropbox/org/toplevel.org")
        ("Appointments" ?a "* Appointment: %?\n%^T\n%i\n  %a" "~/Dropbox/org/toplevel.org")))
(setq remember-annotation-functions '(org-remember-annotation))
(setq remember-handler-functions '(org-remember-handler))
(eval-after-load 'remember
```

[5] http://sachachua.com/blog/2007/12/emacs-getting-things-done-with-org-basic/

```
  '(add-hook 'remember-mode-hook 'org-remember-apply-template))
(global-set-key (kbd "C-c r") 'remember)
```

20 LaTeXCustomization

Nothing yet....

21 Tidying up

Be a nicely-behaved module or "feature":

```
(provide 'init)
```