Jacob Chaffin's Literate Emacs Configuration

Jacob Chaffin

November 19, 2017

Configuration

Preface

About

Literate Programming is a method of writing computer programs where the composition, logic, and structure of the program are optimized for human comprehension. Introduced by Donald Knuth in his 1983 eponymous work, a literate program interoperates source code with macros, commentary, and documentation written in natural language prose. The source code is then extracted in a pre-compilation step known as *tangling*.

Where as a program is traditionally presented in a tree structure, a programmer implementing literate techniques arranges parts and forms the relations of these parts in the order of human logic.

A literate program is then structured like an essay or other work of literature, where ideas are connected in the form of a web rather than the unidirectional order of interpretation that is characteristic of a compiler.

Influences

- Sacha Chua's Emacs Configuration
- Aaron Bedra's Emacs Configuration
- wasamasa/dotemacs
- Optimal Emacs Settings For Org Mode For Literal Programming

Package Management

straight.el

Currently we have a tumultuous relationship.

Use Package

```
(eval-when-compile
    (require 'use-package))
  (require 'diminish)
  (require 'bind-key)
```

Default Packages

```
(require 'cl)
(require 'dash)
```

User Information

These values are initialized with the 'name' and 'email' environment variables, respectively[1].

Emacs uses these variables to fill the mail header when sending emails in emacs, and various third-party packages rely on them for correct behavior.

```
  (setq user-full-name "Jacob Chaffin"
        user-mail-address "jchaffin@ucla.edu")
```

Ensure that programs store emacs information in the cannonical directory.

```
  (setq user-emacs-directory "~/.emacs.d/")
```

Core

Macros

`with-system` This is a simple utility macro that evaluates code depending on operating system type.

Where as the top level `system-type` aliases are intended to simplify conditionals of the *if-then* form or *if-else-then* form, the `with-system` macro is intended for use in structures with a single conditional clause.

[1]Emacs Manual - C.4.1 General Variables

In emacs-lisp, this kind of statement can be expressed with the `when` macro[2].

Credits to stack overflow user gerstmann, who provided this solution in the following stack overflow ticket.

```
(defmacro with-system (type &rest body)
  "If TYPE equals ‘system-type’, evaluate BODY."
  (declare (indent defun))
  ‘(when (eq system-type ’,type)
     ,@body))
```

Functions

Add multiple hooks See Stack Overflow ticket #7398216

```
  (defun add-to-hooks (fun hooks)
          "Add function to hooks"
          (dolist (hook hooks)
           (add-hook hook fun)))
```

Join Strings with Separator

```
(defun join (lst sep)
  (mapconcat ’identity lst sep))
```

Environment

macOS

I work on a macbook, so this block is where I’m loading all of my settings that rely on local filepaths, macOS applications, and external programs.

```
(defconst *is-mac* (eq system-type ’darwin))
```

Computer Name

```
(defun chaffin/computer-name-cmd ()
  (let* ((has-scutil (executable-find "scutil"))
 (scutil-cmd (lambda () (shell-command-to-string "scutil --get ComputerName"))))
    (if has-scutil
```

[2]Emacs Manual - 10.2 Conditionals

```
(replace-regexp-in-string "\n" "" (funcall scutil-cmd))
      nil)))

(defvar computer-name (chaffin/computer-name-cmd))
(defconst *is-hal* (string= computer-name "hal"))
```

Keybindings macOS specific settings[3].

Maps the modifier keys based on personal preferences. Also sets terminal coding system to "utf-8".

```
(setq mac-command-modifier 'super
            mac-option-modifier 'meta
            ns-control-modifier 'control
            ns-function-modifier 'hyper)

(set-terminal-coding-system 'utf-8)
(prefer-coding-system 'utf-8)
```

Reveal in OSX Finder

```
  (use-package reveal-in-osx-finder
    :ensure t
    :bind ("C-c z" . reveal-in-osx-finder))
```

OSX Dictionary Provides an interface to *Dictionary.app* in Emacs.

```
  (use-package osx-dictionary
    :ensure t
    :bind (("C-c C-d" . osx-dictionary-search-word-at-point)
           ("C-c i" . osx-dictionary-search-input)))
```

Exec Path From Shell Emacs relies heavily on shell environment variables.

These variables may not be picked up when launching emacs from a gui on a unix-like operating system.

The exec-path-from-shell package fixes this problem by copying user environment variables from the shell.

[3]EmacsForMacOS

The `exec-path-from-shell` package only works with posix-compliant operating systems. This may or may not include Microsoft Windows[4].

However, the `exec-path-from-shell` instructions recommends loading the package on linux and macOS operating system. I don't use Windows all that often anyways, so that's fine with me.

The `:if` key of `use-package` offers us a really concise way for conditionally loading dependencies.

```
(use-package exec-path-from-shell
  ;; only load `exec-path-from-shell' package on macos and linux.
  :if (memq window-system '(mac ns))
  :ensure t
  :config
  (exec-path-from-shell-initialize)
  (setq exec-path-from-shell-check-startup-files nil))
```

Other Operating Systems

Windows/PC

```
(defconst *is-windows* (eq system-type 'windows))
```

Linux

```
(defconst *is-linux* (eq system-type 'linux))
```

Graphical Interace

There's some packages and commands I only want available when Emacs is running in a graphical interface. Things like image-rendering, large packages, and image rendering.

```
(defconst *is-gui* (display-graphic-p))
```

Custom File

By default, Emacs customizations[5] done through the `customize` interface write to `user-init-file`.

While I usually prefer configuring emacs programmatically, settings that depend on resources outside of this repository, such as org-agenda files, will impact portability and potentially break on other machines.

[4]Quora - Is Windows POSIX compliant?
[5]Emacs Manual - 51.1.4 Saving Customizations

```
(setq custom-file "~/.emacs.d/custom/custom.el")
(load custom-file)
```

Backup Files

This might come back to bite me one day but I never use them.

```
(setq make-backup-files nil)
```

Use Emacs Terminfo

Setting this variable to false forces Emacs to use internal terminfo, rather than the system terminfo.

```
  (setq system-uses-terminfo nil)
```

Org Mode

Org Prettify

These settings subjectively improve the overall viewable-ness of org-mode buffers.

```
  (add-hook #'org-mode-hook (lambda ()
                              (auto-fill-mode)
                              (visual-line-mode)))
(setq org-image-actual-width 400)
  (use-package org-bullets
    :ensure t
    :init
    (add-hook 'org-mode-hook (lambda () (org-bullets-mode 1))))

(use-package toc-org
  :ensure t
  :init
  (add-hook 'org-mode-hook 'toc-org-enable))
```

Improve Legibility

```
  (add-hook #'org-mode-hook (lambda ()
                              (auto-fill-mode)
                              (visual-line-mode)))
```

Image Size Make images compact.

```
(setq org-image-actual-width 400)
```

Org Bullets Use UTF-8 Bullets for Org-mode headings.

```
  (use-package org-bullets
    :ensure t
    :init
    (add-hook 'org-mode-hook (lambda () (org-bullets-mode 1))))
```

Org Beautify Theme

```
  (use-package org-beautify-theme
    :ensure nil
    :defines org-beautify-theme-use-box-hack
    :config
    (setq org-beautify-theme-use-box-hack nil)
    (add-to-list 'custom-enabled-themes 'org-beautify)
    (load-theme 'org-beautify t))
```

TOC Org

```
(use-package toc-org
  :ensure t
  :init
  (add-hook 'org-mode-hook 'toc-org-enable))
```

Org Utilities

```
  (use-package org-agenda
    :recipe org
    :defer-install t
    :bind (("C-c a" . org-agenda)))
  (use-package org-pomodoro
    :ensure t
    :bind (:map org-mode-map
                ("C-c M-RET p" . org-pomodoro))
    :config
    (setq org-pomodoro-audio-player "afplay"
          org-pomodoro-tick-sound
          (expand-file-name "~/.emacs.d/resources/clock-ticking-2.wav")
          ;; Start Settings
          org-pomodoro-start-sound-p t ;; enable starting sound
          org-pomodoro-start-sound-args "--volume 0.08"
          org-pomodoro-start-sound
          (expand-file-name "~/.emacs.d/resources/Victory.wav")
          ;; Finished Settings
          org-pomodoro-finished-sound-args "--volume 0.2"
          org-pomodoro-finished-sound
          (expand-file-name "~/.emacs.d/resources/Waves.wav")
          ;; Short Break Settings
          org-pomodoro-short-break-length 5
          org-pomodoro-short-break-sound-args "--volume 0.2"
          org-pomodoro-short-break-sound org-pomodoro-finished-sound
          ;; Long Break Settings
          org-pomodoro-long-break-length 15
          org-pomodoro-long-break-sound-args "--volume 0.2"
          org-pomodoro-long-break-sound
          (expand-file-name "~/.emacs.d/resources/Complete.wav")))
  (use-package org-ref
    :if *is-mac*
    :ensure t
    :config
    (progn
      (setq org-ref-bibliography-notes "~/Dropbox/org/papers/notes.org"
            org-ref-default-bibliography '("~/Dropbox/org/papers/references.bib")
            org-ref-pdf-directory "~/Dropbox/org/papers/pdfs/")
      (require 'org-ref-pdf)
      (require 'org-ref-latex)
      (require 'org-ref-reftex)
      (require 'org-ref-utils)
      (require 'org-ref-url-utils)
      (require 'org-ref-pubmed)
      (require 'org-ref-scopus)
      (require 'org-ref-scifinder)
      (require 'org-ref-citeproc)
      (require 'org-ref-sci-id)
      (require 'org-ref-isbn)
      (require 'org-ref-wos)
```

```
      (require 'org-ref-worldcat)
      (require 'x2bib)))
(setq org-ref-completion-library 'org-ref-ivy-cite)
(require 'org-ref)
(require 'org-ref-ivy)
(require 'org-ref-ivy-cite)
(defun chaffin/org-ref-open-pdf-at-point ()
  "Open the pdf for bibtex key under point if it exists."
  (interactive)
  (let* ((results (org-ref-get-bibtex-key-and-file))
         (key (car results))
         (pdf-file (funcall org-ref-get-pdf-filename-function key)))
    (if (file-exists-p pdf-file)
        (find-file pdf-file)
      (message "No PDF found for %s" key))))

(setq org-ref-open-pdf-function 'chaffin/org-ref-open-pdf-at-point)

(require 'org-id)
(setq org-id-link-to-org-use-id 'create-if-interactive-and-no-custom-id)
(defun chaffin/org-custom-id-get (&optional pom create prefix)
  "Get the CUSTOM_ID property of the entry at point-or-marker POM.
   If POM is nil, refer to the entry at point. If the entry does
   not have an CUSTOM_ID, the function returns nil. However, when
   CREATE is non nil, create a CUSTOM_ID if none is present
   already. PREFIX will be passed through to `org-id-new'. In any
   case, the CUSTOM_ID of the entry is returned."
  (interactive)
  (org-with-point-at pom
    (let ((id (org-entry-get nil "CUSTOM_ID")))
      (cond
       ((and id (stringp id) (string-match "\\S-" id))
        id)
       (create
        (setq id (org-id-new (concat prefix "h")))
        (org-entry-put pom "CUSTOM_ID" id)
        (org-id-add-location id (buffer-file-name (buffer-base-buffer)))
        id)))))
```

Org Agenda

```
(use-package org-agenda
  :recipe org
  :defer-install t
  :bind (("C-c a" . org-agenda)))
```

Org Pomodoro

```
(use-package org-pomodoro
  :ensure t
  :bind (:map org-mode-map
              ("C-c M-RET p" . org-pomodoro))
  :config
  (setq org-pomodoro-audio-player "afplay"
        org-pomodoro-tick-sound
        (expand-file-name "~/.emacs.d/resources/clock-ticking-2.wav")
        ;; Start Settings
        org-pomodoro-start-sound-p t ;; enable starting sound
        org-pomodoro-start-sound-args "--volume 0.08"
        org-pomodoro-start-sound
        (expand-file-name "~/.emacs.d/resources/Victory.wav")
        ;; Finished Settings
        org-pomodoro-finished-sound-args "--volume 0.2"
        org-pomodoro-finished-sound
        (expand-file-name "~/.emacs.d/resources/Waves.wav")
        ;; Short Break Settings
        org-pomodoro-short-break-length 5
        org-pomodoro-short-break-sound-args "--volume 0.2"
        org-pomodoro-short-break-sound org-pomodoro-finished-sound
        ;; Long Break Settings
        org-pomodoro-long-break-length 15
        org-pomodoro-long-break-sound-args "--volume 0.2"
        org-pomodoro-long-break-sound
        (expand-file-name "~/.emacs.d/resources/Complete.wav")))
```

Org Ref

```
(use-package org-ref
  :if *is-mac*
  :ensure t
  :config
  (progn
    (setq org-ref-bibliography-notes "~/Dropbox/org/papers/notes.org"
          org-ref-default-bibliography '("~/Dropbox/org/papers/references.bib")
          org-ref-pdf-directory "~/Dropbox/org/papers/pdfs/")
    (require 'org-ref-pdf)
    (require 'org-ref-latex)
    (require 'org-ref-reftex)
    (require 'org-ref-utils)
    (require 'org-ref-url-utils)
    (require 'org-ref-pubmed)
    (require 'org-ref-scopus)
    (require 'org-ref-scifinder)
    (require 'org-ref-citeproc)
    (require 'org-ref-sci-id)
    (require 'org-ref-isbn)
    (require 'org-ref-wos)
    (require 'org-ref-worldcat)
    (require 'x2bib)))
```

Org Ref Ivy

```
(setq org-ref-completion-library 'org-ref-ivy-cite)
(require 'org-ref)
(require 'org-ref-ivy)
(require 'org-ref-ivy-cite)
```

Open Pdf at point See Using Doc View or PDF Tools section of `org-ref` documentation.

```
(defun chaffin/org-ref-open-pdf-at-point ()
  "Open the pdf for bibtex key under point if it exists."
  (interactive)
  (let* ((results (org-ref-get-bibtex-key-and-file))
         (key (car results))
         (pdf-file (funcall org-ref-get-pdf-filename-function key)))
    (if (file-exists-p pdf-file)
        (find-file pdf-file)
      (message "No PDF found for %s" key))))
```

```
(setq org-ref-open-pdf-function 'chaffin/org-ref-open-pdf-at-point)
```

Org ID

Custom ID

```
(require 'org-id)
(setq org-id-link-to-org-use-id 'create-if-interactive-and-no-custom-id)
```

- Get Custom ID

  From Emacs Org Mode Generate Ids blogpost:

  ```
  (defun chaffin/org-custom-id-get (&optional pom create prefix)
    "Get the CUSTOM_ID property of the entry at point-or-marker POM.
     If POM is nil, refer to the entry at point. If the entry does
     not have an CUSTOM_ID, the function returns nil. However, when
     CREATE is non nil, create a CUSTOM_ID if none is present
     already. PREFIX will be passed through to 'org-id-new'. In any
     case, the CUSTOM_ID of the entry is returned."
    (interactive)
    (org-with-point-at pom
      (let ((id (org-entry-get nil "CUSTOM_ID")))
        (cond
         ((and id (stringp id) (string-match "\\S-" id))
          id)
         (create
          (setq id (org-id-new (concat prefix "h")))
          (org-entry-put pom "CUSTOM_ID" id)
          (org-id-add-location id (buffer-file-name (buffer-base-buffer)))
          id)))))
  ```

Org PDF

Pdf Tools

```
(use-package pdf-tools
  :ensure t
  :defer t
  :config
  (pdf-tools-install))
```

Org PDF View

```
  (use-package org-pdfview
    :after pdf-tools
    :ensure t
    :mode (("\\.pdf\\'" . pdf-view-mode)))
```

Now we can open pdf files with `org-pdfview`[6]:

```
(with-eval-after-load 'org-pdfview
  (add-to-list 'org-file-apps '("\\.pdf\\'" . (lambda (file link) (org-pdfview-open li
```

Org Export

LaTeX Backend

Latexmk Latexmk automates the proces of building LaTeX documents to pdf.

It can be done through the `org-mode` latex export dispatcher in a single command:

```
(setq org-latex-pdf-process
    '("latexmk -pdflatex='pdflatex -interaction nonstopmode' -synctex=1 -pdf -bibtex -
```

Prefer user provided labeling system

```
(setq org-latex-prefer-user-labels t)
```

Org LaTeX Default Packages

```
<<org-export-latex-default-packages-natbib>>
```

- Natbib

  Add natbib package, as it's the easiest way to get BibTeX support through org-ref and other tools for LaTeX in Emacs.

  ```
  (add-to-list 'org-latex-default-packages-alist '("" "natbib" "") t)
  ```

[6]See Note taking with pdf-tools

Custom Classes

- Org Per File Class Adds a class for exporting to pdf using latex backend without importing the default `ox-latex` packages. This allows the export settings of a particular file to be completely insulated from most external configuration.

```
(add-to-list 'org-latex-classes
     '("per-file-class"
       "\\documentclass{scrartcl}
       [NO-DEFAULT-PACKAGES]
       [EXTRA]"
       ("\\section{%s}" . "\\section*{%s}")
       ("\\subsection{%s}" . "\\subsection*{%s}")
       ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
       ("\\paragraph{%s}" . "\\paragraph*{%s}")
       ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
```

GitHub Flavored Markdown

```
(use-package ox-gfm
  :ensure t
  :init
  (with-eval-after-load 'org-mode
    (require 'ox-gfm)))
```

Org YouTube From Endless Parentheses blogpost:
Embed YouTube Links in iframe.

```
(defvar yt-iframe-format
  ;; You may want to change your width and height.
  (concat "<iframe width=\"440\""
          " height=\"335\""
          " src=\"https://www.youtube.com/embed/%s\""
          " frameborder=\"0\""
          " allowfullscreen>%s</iframe>"))

(org-add-link-type
 "yt"
 (lambda (handle)
   (browse-url
    (concat "https://www.youtube.com/embed/"
            handle)))
 (lambda (path desc backend)
   (cl-case backend
     (html (format yt-iframe-format
                   path (or desc "")))
     (latex (format "\href{%s}{%s}"
                    path (or desc "video"))))))
```

Ox Extra Ignores headlines tagged "ignore". Unlike "noexport", the contents and subtrees of the ignored headlines will be retained.

```
(require 'ox-extra)
(ox-extras-activate '(ignore-headlines))
```

(Better) Defaults

Better Utilities

Dired+ Adds extensions and functionality to dired mode.

```
  (use-package dired+
    :ensure t)
```

Undo Tree

```
  (use-package undo-tree
    :ensure t
    :init
    (global-undo-tree-mode))
```

Projectile

```
  (use-package projectile
    :ensure t
    :config
    (projectile-mode))
```

Restart Emacs The restart-emacs package allows quickly rebooting Emacs from within Emacs.

```
(use-package restart-emacs
  :ensure t
  :bind (("C-x C-r" . restart-emacs)))
```

Popwin

```
(use-package popwin
  :ensure t
  :config (popwin-mode 1))
```

Better Display

Page Break Lines Global minor-mode that turns `^L` form feed characters into horizontal line rules.

```
(use-package page-break-lines
  :ensure t
  :init
  (global-page-break-lines-mode))
```

Splash Screen Replacement. The default splash screen is great when you're starting out, but it's more so an annoyance than anything else once you know you're around.

Enable Emojis for org tags in dashboard agenda widget

```
(use-package emojify
  :ensure t
  :init (global-emojify-mode))
```

Emacs Dashboard Dashboard is a highly customizable splash screen replacement library used in the popular spacemacs framework. It's a nice way of consolidating any combination of tasks, agenda items, bookmarks, and pretty much any other enumerable list that one may use in the wacky world of Emacs.

```
(use-package dashboard
  :ensure t
  :init
  (with-eval-after-load 'page-break-lines
    (if (not (global-page-break-lines-mode))
        (global-page-break-lines-mode)))
  :config
  (dashboard-setup-startup-hook))

(progn
  (add-to-list 'dashboard-items '(agenda) t)
  (setq dashboard-banner-logo-title "Welcome Back, MasterChaff"
        dashboard-items '(( agenda . 10)
                          ( projects . 5)
                          ( recents . 5)
                          ( bookmarks . 5))))
```

Inhibit Scratch Buffer

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

Menu Bar, Tool Bar, Scroll Bar Disable scroll bars and tool bar on all system types.

On macOS, the menu bar is integrated into the UI.

Disabling it will just empty the menu tab options for Emacs.app, so we'll leave it there.

```
(scroll-bar-mode -1)
(tool-bar-mode -1)
(unless (eq system-type 'darwin)
(menu-bar-mode -1))
```

Truncate Lines Not sure this is doing anything. . .

```
(setq-default truncate-lines nil)
```

Better Garbage Collection

Consider the following from the documentation:

> By binding this temporarily to a large number, you can effectively prevent garbage collection during a part of the program.

When I first read how the default garbage collection interval in Emacs is notoriously low, I added an arbitrary number of zeros to the default value and called it a day. However, because I'm writing this monolithic configuration and making a lot of mistakes in the process, I've had to start Emacs with essentially its default settings pretty frequently and I've noticed the lag time I occasionally experience when searching long documents is essentially nonexistent in vanilla Emacs. After reading this blogpost, it seems that jacking up the GC interval may actually be the *cause* of the lagtime rather than contributing to the solution.

```
(defun chaffin/rev-up-gc ()
  (setq gc-cons-threshold most-positive-fixnum))

(defun chaffin/rev-down-gc ()
  (setq gc-cons-threshold 800000))

(add-hook 'minibuffer-setup-hook #'chaffin/rev-up-gc)
(add-hook 'minibuffer-exit-hook 'chaffin/rev-down-gc)
```

Better Encryption

GnuTLS

- See wasamasa/dotfiles

```
(setq gnutls-min-prime-bits 4096)
```

Use GPG2 Set GPG program to 'gpg2'.

```
(when *is-mac*
  (setq epg-gpg-program "gpg2"))
```

Disable External Pin Entry Switching between Emacs and an external tools is annoying.

By default, decrypting gpg files in Emacs will result in the pin entry window being launched from the terminal session.

By disabling the agent info, we can force Emacs to handle this internally[7].

```
(setenv "GPG_AGENT_INFO" nil)
```

Or so I thought…

Internal Pinentry Problem and Solution While I couldn't figure out how to get Emacs to handle gpg pinentry internally, I was able to still find a satisfactory solution using the `pinentry-mac` tool.

Note that this solution requires macOS and using gpg2 for encryption.

See ticket #1437 from the Homebrew/homebrew-core repository.

```
brew install pinentry-mac
echo "pinentry-program /usr/local/bin/pinentry-mac" >> ~/.gnupg/gpg-agent.conf
killall gpg-agent
```

Better Commands

Alias Yes Or No

```
(defalias 'yes-or-no-p 'y-or-n-p)
```

Editing

Indentation

Tabs are the bane of humanity[8]. Don't @ me.

```
(setq tab-width 2
      indent-tabs-mode nil)
```

[7]Mastering Emacs - Keeping Secrets in Emacs GnuPG Auth Sources
[8]Emacs Wiki - Tabs Are Evil

`highlight-indent-guides` Highlight Indent Guides sublime-like indentation guides.

Commented out because of bug that leaves a trail of solid white line marks on the indent guide overlay.

```
(use-package highlight-indent-guides
   :ensure t
   :init
   (add-hook 'prog-mode-hook 'highlight-indent-guides-mode)
   :config
   (setq highlight-indent-guides-method 'character))
```

YASnippet

YASnippet is a template system based off the TextMate snippet syntax.

Let's begin by creating a variable for our personal snippets directory.

```
(setq user-snippets-dir (concat user-emacs-directory "snippets"))
```

After installation and enabling the package, add the personal snippets directory to the list of directories where YASnippet should look for snippets.

```
(use-package yasnippet
  :ensure t
  :init
  (yas-global-mode 1)
  :config
  (push 'user-snippets-dir yas-snippet-dirs))
```

YASnippet can also be used as a non-global minor mode on a per-buffer basis.

Invoking `yas-reload-all` will load the snippet tables, and then calling `yas-minor-mode` from the major mode hooks will load the snippets corresponding to the major mode of the current buffer mode.

```
(yas-reload-all)
(add-hook 'prog-mode-hook #'yas-minor-mode)
```

Flycheck

On the fly syntax highlighting.

```
(use-package flycheck
  :defer-install t
  :init
  (setq flycheck-global-modes nil)
  :config
    (setq-default flycheck-disabled-checkers '(emacs-lisp-checkdoc)
                  flycheck-emacs-lisp-load-path 'inherit)
           (use-package flycheck-color-mode-line
      :ensure t
      :init
      (add-hook 'flycheck-mode 'flycheck-color-mode-line-mode)))
```

Flycheck Color Mode Line Colors the modeline according to current Flycheck state of buffer.

```
      (use-package flycheck-color-mode-line
:ensure t
:init
(add-hook 'flycheck-mode 'flycheck-color-mode-line-mode))
```

Flycheck Package Flycheck Package requires `package.el` to be enabled, so it's incompatible with `straight.el`.

```
(use-package flycheck-package
  :ensure t
  :init
  (eval-after-load 'flycheck
    '(flycheck-package-setup)))
```

Flycheck in Org Special Edit Buffers

```
(defadvice org-edit-src-code (around set-buffer-file-name activate compile)
  (let ((file-name (buffer-file-name))) ;; (1)
    ad-do-it                            ;; (2)
    ;; (3)
    (setq buffer-file-name file-name)))
```

Company

Emacs has two popular packages for code completion – autocomplete and company. This reddit thread was enough for me to go with company.

If you need more convincing, company-mode/company-mode#68 offers a comprehensive discussion on the two.

The ticket is from the ‘company-mode‘ repository, so there’s probably some bias there, but company-mode hasn’t provided any reason for me reconsider my choice.

```
(global-company-mode)
(setq company-tooltip-limit 20
      company-tooltip-align-annotations t
      company-idle-delay .3
      company-begin-commands '(self-insert-command))
```

Company Statistics Company statistics uses a persisent store of completions to rank the top candidates for completion.

```
  (use-package company-statistics
    :ensure t
    :config
    ;; Alternatively,
    ;; (company-statistics-mode)
    (add-hook 'after-init-hook 'company-statistics-mode))
```

Company Quick Help Company Quick Help emulates `autocomplete` documentation-on-idle behavior, but using the less-buggy `pos-tip` package rather than `popup-el`.

```
  (use-package company-quickhelp
    :defer t
    :commands (company-quickhelp-manual-begin)
    :bind
    (:map company-active-map
          ("C-c h" . company-quickhelp-manual-begin))
    :config
    (company-quickhelp-mode 1))
```

Company Dict

```
(use-package company-dict
 :ensure t
 :init
 (add-to-list 'company-backends 'company-dict)
 :config
 (setq company-dict-enable-fuzzy t
       company-dict-enable-yasnippet t))
```

Utilities

Image+

Image+ provides extensions for image file manipulation in Emacs.

```
(use-package image+
  :ensure t
  :if *is-gui*
  :after image
  :config
  (eval-after-load 'image+
    `(when (require 'hydra nil t)
       (defhydra imagex-sticky-binding (global-map "C-x C-l")
         "Manipulating image"
         ("+" imagex-sticky-zoom-in "zoom in")
         ("-" imagex-sticky-zoom-out "zoom out")
         ("M" imagex-sticky-maximize "maximize")
         ("O" imagex-sticky-restore-original "resoure orginal")
         ("S" imagex-sticky-save-image "save file")
         ("r" imagex-sticky-rotate-right "rotate right")
         ("l" imagex-sticky-rotate-left "rotate left")))))
```

Ivy

Ivy Config Ivy is a completion and selection framework in the same vein as helm. It doesn't have the same kind of ecosystem or interopability, but its easy to configure, offers a minimalistic interface, and is every bit as good of a completion tool as helm is, if not better.

I prefer the default regex matcher, but if you want fuzzy matching as a fallback or replacement checkout this article on better fuzzmatching support with ivy.

```
(use-package ivy
  :config
  (ivy-mode 1)
  (setq ivy-use-virtual-buffers t
        ivy-initial-inputs-alist nil
        ivy-re-builders-alist
'((t . ivy--regex-plus)))
    (use-package ivy-hydra
      :ensure hydra)
  (use-package historian
    :ensure t)
  (use-package ivy-historian
    :after historian
    :ensure t))
```

Ivy Hydra

```
(use-package ivy-hydra
  :ensure hydra)
```

Ivy Historian

```
(use-package ivy-historian
  :after historian
  :ensure t)
```

Counsel

```
       (use-package counsel
         :ensure t
         :bind
         (("C-c C-r" . ivy-resume)
          ("C-`" . ivy-avy)
         ("M-x" . counsel-M-x)
         ("M-y" . counsel-yank-pop)
         ("C-x C-f" . counsel-find-file)
         ("<f1> f" . counsel-describe-function)
         ("<f1> v" . counsel-describe-variable)
         ("<f1> l" . counsel-load-library)
         ("<f2> i" . counsel-info-lookup-symbol)
         ("<f2> u" . counsel-unicode-char)
```

```
("C-c g" . counsel-git)
("C-c j" . counsel-git-grep)
("C-c k" . counsel-ag)
("C-x l" . counsel-locate)
("C-S-o" . counsel-rhythmbox)
:map read-expression-map
("C-r" . counsel-expression-history))
:init
  (progn
      (use-package ivy
        :config
        (ivy-mode 1)
        (setq ivy-use-virtual-buffers t
              ivy-initial-inputs-alist nil
              ivy-re-builders-alist
       '((t . ivy--regex-plus)))
          (use-package ivy-hydra
            :ensure hydra)
        (use-package historian
          :ensure t)
        (use-package ivy-historian
          :after historian
          :ensure t))
      (use-package swiper
      :bind
      (("\C-s" . swiper))
      :init
        (use-package avy
          :ensure t))
      (use-package counsel-projectile
        :ensure t
        :init
        (progn
          (counsel-projectile-on)))
       (use-package counsel-osx-app
         :if *is-mac*
         :ensure t
         :bind (("C-c o a" . counsel-osx-app)))
      (use-package counsel-dash
        :if *is-mac*
```

```
                  :ensure t
                  :init (defun counsel-dash-at-point ()
                          "Counsel dash with selected point."
                          (interactive)
                          (counsel-dash
                           (if (use-region-p)
                               (buffer-substring-no-properties
                                (region-beginning)
                                (region-end))
                             (substring-no-properties (or (thing-at-point 'symbol) ""))
                  :config
                  (setq counsel-dash-docsets-path
                        (expand-file-name "~/Library/Application\sSupport/Dash/DocSets"
                (use-package smex
                  :ensure t
                  :init (setq-default smex-history-length 32))))
```

Swiper

```
(use-package swiper
:bind
(("\C-s" . swiper))
:init
  (use-package avy
    :ensure t))
```

Swiper Avy

```
(use-package avy
  :ensure t)
```

Counsel-Projectile Counsel Projectile provides a project management interface via ivy and friends.

```
(use-package counsel-projectile
  :ensure t
  :init
  (progn
    (counsel-projectile-on)))
```

Smex

```
(use-package smex
  :ensure t
  :init (setq-default smex-history-length 32))
```

Counsel OSX App

```
 (use-package counsel-osx-app
   :if *is-mac*
   :ensure t
   :bind (("C-c o a" . counsel-osx-app)))
```

Counsel Dash

```
(use-package counsel-dash
  :if *is-mac*
  :ensure t
  :init (defun counsel-dash-at-point ()
          "Counsel dash with selected point."
          (interactive)
          (counsel-dash
           (if (use-region-p)
               (buffer-substring-no-properties
                (region-beginning)
                (region-end))
             (substring-no-properties (or (thing-at-point 'symbol) "")))))
  :config
  (setq counsel-dash-docsets-path
        (expand-file-name "~/Library/Application\sSupport/Dash/DocSets")))
```

Deft

Deft is a notetaking application for Emacs.

```
(use-package deft
  :ensure t
  :bind ("C-x C-n" . deft)
  :config
  (setq deft-extensions '("org")
        deft-directory "~/Dropbox/org/notes"
```

```
          deft-use-filename-as-title t
          deft-default-extension "org"))
```

Wakatime

```
  (use-package wakatime-mode
    :if (and *is-mac* (or (string= (downcase computer-name) "hal") (string= (downcase c
    :ensure t
    :init
    (add-hook 'prog-mode-hook 'wakatime-mode)
    :config
    (progn
      (setq wakatime-cli-path
            (expand-file-name "~/.local/lib/python3.6/site-packages/wakatime/cli.py")
            wakatime-python-bin
            (expand-file-name "~/.pyenv/shims/python"))

      (defun wakatime-dashboard ()
        (interactive)
        (browse-url "https://wakatime.com/dashboard"))))
```

User Interface

Cursor

Vertical Bar Set the cursor to a bar. The default is too thin for my liking. Set the width to 4px. Also remove the cursor in inactive windows.

```
(setq-default cursor-type '(bar . 4)
              cursor-in-non-selected-windows 'nil
              x-stretch-cursor t
              line-spacing 2)
```

Disable Blink Ultimately, I'd like to set a longer blink interval, like the "phase" `caret_style` setting in Sublime Text.

```
(blink-cursor-mode -1)
```

Smart Cursor Color

```
(use-package smart-cursor-color
  :ensure t
  :config
  (smart-cursor-color-mode +1))
```

Theme

Enable Custom Themes This disables Emacs asking questions about loading a new theme.

```
(setq custom-safe-themes t)
```

Zenburn Theme

```
(use-package zenburn-theme
  :ensure t
  :config (load-theme 'zenburn))
```

Load Themes Zenburn theme has to be loaded before Org Beautify Theme to preserve compatibility.

Modeline

Display Time Show the time in the modeline.

```
(display-time-mode 1)
```

Smart-Mode-Line

```
(use-package smart-mode-line
  :ensure t
  :init
  (smart-mode-line-enable)
  :config
  (setq sml/mode-width 0
        sml/name-width 20
        sml/not-confirm-load-theme t)
  (setf rm-blacklist "")
  (sml/setup))
```

Mode Icons

```
(use-package mode-icons
  :ensure t
  :if *is-gui*
  :after smart-mode-line
  :config
  (mode-icons-mode))
```

All The Icons

All The Icons is a utility package for icons in Emacs.

```
(use-package all-the-icons
  :if *is-gui*
  :ensure t
  :init
  (unless (straight-check-package "all-the-icons")
    (all-the-icons-install-fonts)))

(use-package all-the-icons-ivy
    :after all-the-icons ivy
    :ensure t
    :if *is-gui*
    :init
    (all-the-icons-ivy-setup))

(use-package all-the-icons-dired
  :if *is-gui*
  :ensure t
  :config
  (add-hook 'dired-mode-hook 'all-the-icons-dired-mode))
```

Terminal

Multi-Term

```
(use-package multi-term
       :ensure t
       :bind
       (("C-c M-RET t" . multi-term)
("C-c M-RET p" . multi-term-prev)
("C-c M-RET n" . multi-term-next)
("C-c M-RET o" . multi-term-dedicated-toggle))
       :config
       (progn
 (setq multi-term-program
     (if (string= shell-file-name "/bin/sh")
 "/bin/bash"
       shell-file-name))))
```

Frame Font

Use the default monospaced font for each operating system.

```
 (cond (*is-linux*
  (set-frame-font "Ubuntu Mono 12" nil t))
 (*is-windows*
  (set-frame-font "Lucida Sans Typewriter 12" nil t))
 ((eq system-type 'darwin)
  (set-frame-font "SF Mono 12" nil t))
 (t
  (set-frame-font "Menlo 12" nil t)))
```

Sunshine

```
  (use-package sunshine
    :ensure t
    :config
    (setq sunshine-location "90024,USA"))
```

Theme Changer

```
  (use-package theme-changer
    :ensure t
    :config
    (change-theme 'zenburn-theme 'anti-zenburn-theme))
```

Emojify

```
  (use-package emojify
    :ensure t
    :init (global-emojify-mode))
```

Version Control

Magit

Magit describes itself as one of two git porcelains, the other being git itself.

A git porcelain is jargon for a program that features a user-friendly vcs interface, as opposed to lower-level scripting commands.

It's not a vitrified ceramic commonly used for decorative tableware. Magit would not be very good at that.

As a git client though, magit is awesome.

```
(use-package magit
  :ensure t
  :bind (("C-c v b" . magit-blame)
         ("C-c v C" . magit-clone)
         ("C-c v c" . magit-checkout)
         ("C-c v i" . magit-init)
         ("C-c v m" . magit-merge)
         ("C-c v l" . magit-log-buffer-file)
         ("C-c v p" . magit-pull)
         ("C-c v P" . magit-push)
         ("C-c v v" . magit-status))
  :config (setq magit-save-repository-buffers 'dontask))
```

Magithub Magithub offers an interface to github to complement magit.

```
(use-package magithub
  :after magit
  :ensure t
  :commands magithub-dispatch-popup
  :bind (:map magit-status-mode-map
              ("@" . magithub-dispatch-popup))
  :config
  (progn
    (magithub-feature-autoinject t)))
```

gist.el

Emacs integration for gist.github.com.

Gist requires generating a personal access token with `gist` scope, and optionally `user` and `repo` scopes.

```
(use-package gist
  :ensure t
  :bind (("C-c C-g l" . gist-list)
         ("C-c C-g r" . gist-region)
         ("C-c C-g b" . gist-buffer)
         ("C-c C-g p" . gist-buffer-private)
         ("C-c C-g B" . gist-region-or-buffer)
         ("C-c C-g P" . gist-region-or-buffer-private)))
```

git-timemachine

Travel back in time (to your last commit).

```
(use-package git-timemachine
  :ensure t
  :bind
  ("C-x v t" . git-timemachine-toggle)
  :config
  (setq git-timemachine-abbreviation-length 7))
```

git-messenger

Pop-up feature for viewing the last git commit.

```
(use-package git-messenger
  :ensure t
  :bind
  (("C-c v m" . git-messenger:popup-message)))
```

git modes

```
(use-package git-modes
  :ensure t)
```

Web Browsing

osx-browse

This library provides several useful commands for using the Google Chrome, Safari, and Firefox web browsers on macOS.

```
(use-package osx-browse
  :ensure t
  :if *is-mac*
  :config
  (osx-browse-mode 1))
```

Email

Mu

```
(use-package mu4e
  :if *is-mac*
  :load-path "/usr/local/Cellar/mu/0.9.18_1/share/emacs/site-lisp/mu/mu4e"
  :config
  (progn
    (setq mu4e-maildir (expand-file-name "~/.mail")
          mu4e-context-policy 'pick-first
          mu4e-compose-policy nil
          mu4e-get-mail-command "offlineimap -q -u quiet"
          mu4e-show-images t
          mu4e-show-addresses t)

     ;; smtp settings
     (setq mu4e-send-mail-function 'smtp-mail-send-it
     mu4e-default-smtp-server "smtp.gmail.com"
     smtpmail-smtp-service 587
     smtp-mail-smtp-stream-type 'ssl)


     (setq mu4e-contexts
           `( ,(make-mu4e-context
                :name "private"
                :match-func (lambda (msg)
                              (when msg
                                mu4e-message-contact-field-matches msg
                                :to "jchaffin57@gmail.com"))
                :leave-func (lambda ()
                              (mu4e-message "Leaving Gmail Account"))
                :vars '( (mu4e-reply-to-address "jchaffin@ucla.edu")
                         ( user-mail-address . "jchaffin57@gmail.com" )
                         ( user-full-name . "Jacob Chaffin" )
                         ( mu4e-drafts-folder . "/private/[Gmail].Drafts" )
                         ( mu4e-sent-folder . "/private/[Gmail].Trash" )
                         ( mu4e-refile-folder . "/archived")
                         ( mu4e-compose-signature .
                                                  (concat
                                                   "Jacob Chaffin\n"
                                                   "UCLA 19\n"
                                                   "Linguistics and Computer Science
                                                   "jchaffin@ucla.edu\n"))))

              ,(make-mu4e-context
                :name "school"
                :enter-func (lambda () (mu4e-message "Switching to UCLA Mail"))
                :leave-func (lambda () (mu4e-message "Leaving UCLA Mail"))
                :match-func  (lambda (msg)
                               (when msg
                                 (mu4e-message-contact-field-matches msg
                                                                     :to (or "jchaff
                :vars '( (user-mail-address . "jchaffin@ucla.edu" )
                         (user-full-name . "Jacob Chaffin" )
                         (mu4e-compose-signature . (concat
                                                    "Jacob Chaffin\n"
                                                    "UCLA 19\n"
                                                    "Linguistics and Computer Scienc
                                                    "jchaffin@ucla.edu\n"
                                                    "(650)-380-3288\n"))))))))
```

Programming Languages

Lisp

Parens

- Paredit and Smartparens

Paredit We could use `:init` key to hook the `enable-paredit-mode` function to each of the implementing languages like is done here, but I think adding the hook in the configuration block of the programming language is easier to follow and offers more meaningful semantics.

```
(use-package paredit
  :ensure t
  :diminish paredit-mode
  :config
  (use-package eldoc
    :ensure t
    :config
    (eldoc-add-command
     'paredit-backward-delete
     'paredit-close-round))
  (autoload 'enable-paredit-mode "paredit" "Turn on pseudo-structural editing of Lis
```

Smartparens Like paredit, smartparens is a minor-mode for managing parens pairs. However, it also offers support for curly brackets in JavaScript objects, angle brackets in HTML, and most other major programming languages. I think I the "delete-on-command" behavior of paredit for lisp programming, but in languages where locating unmatched pairs is less comparable to searching for a needle in a haystack, smartparens are a great productivity tool.

```
(use-package smartparens
  :ensure t
  :init
  (require 'smartparens-config))
```

Hlsexp Minor mode to highlight s-expresion.

```
(use-package hl-sexp
  :ensure t)
```

Common-Lisp

Configuration for emacs-lisp.

```
(use-package lisp-mode
  :after paredit
  :config
  (add-hook 'lisp-mode-hook #'paredit-mode)
  (add-hook 'lisp-mode-hook #'hl-sexp-mode)
  (add-hook 'emacs-lisp-mode-hook #'paredit-mode)
  (add-hook 'emacs-lisp-mode-hook #'hl-sexp-mode))
```

Slime SLIME is The Superior Lisp Interaction Mode for Emacs.

```
(use-package slime
  :commands slime
  :defines (slime-complete-symbol*-fancy slime-completion-at-point-functions)
  :ensure t
  :if *is-mac*
  :init
  (progn
    (setq slime-contribs '(slime-asdf
                           slime-fancy
                           slime-indentation
                           slime-sbcl-exts
                           slime-scratch)
          inferior-lisp-program "sbcl"
          ;; enable fuzzy matching in code buffer and SLIME REPL
          slime-complete-symbol*-fancy t
          slime-completion-at-point-functions 'slime-fuzzy-complete-symbol)
    (defun slime-disable-smartparens ()
      (smartparents-strict-mode -1)
      (turn-off-smartparens-mode))
    (add-hook 'slime-repl-mode-hook #'slime-disable-smartparens)))
```

Clojure

Clojure Mode (repository) Provides key bindings and code colorization for Clojure(Script).

```
(use-package clojure-mode
  :ensure t
  :mode (("\\.edn$" . clojure-mode)
         ("\\.cljs$" . clojurescript-mode)
         ("\\.cljx$" . clojurex-mode)
         ("\\.cljsc$" . clojurec-mode))
  :config
  (add-hook 'clojure-mode-hook #'enable-paredit-mode)
    (use-package cljsbuild-mode
      :ensure t
      :init
      (add-to-hooks #'cljsbuild-mode '(clojure-mode clojurescript-mode)))
    (use-package elein
      :ensure t))
```

ClojureScript

Lein Cljsbuild Minor mode offering `lein cljsbuild` commands for the Leiningen plugin.

```
(use-package cljsbuild-mode
  :ensure t
  :init
  (add-to-hooks #'cljsbuild-mode '(clojure-mode clojurescript-mode)))
```

elein Elein rovides support for leiningen commands in Emacs.

```
(use-package elein
  :ensure t)
```

Clojure Mode Extra Font Locking Additional syntax highlighting for `clojure-mode`.

```
(use-package clojure-mode-extra-font-locking
  :ensure t)
```

Cider (repository) Provides integration with a Clojure repl.

```
(use-package cider
  :ensure t
  :after company
  :config
  (setq cider-repl-history-file "~/.emacs.d/cider-history"
        cider-repl-use-clojure-font-lock t
        cider-repl-result-prefix ";; => "
        cider-repl-wrap-history t
        cider-repl-history-size 3000
        cider-show-error-buffer nil
        nrepl-hide-special-buffers t)
  (add-hook 'cider-mode-hook #'eldoc-mode)
  (add-hook 'cider-mode-hook #'company-mode)
  (add-hook 'cider-repl-mode-hook #'cider-company-enable-fuzzy-completion)
```

```
    (add-hook 'cider-mode-hook #'cider-company-enable-fuzzy-completion)
    (add-hook 'cider-repl-mode-hook #'company-mode)
    (add-hook 'cider-repl-mode-hook #'subword-mode)
    (add-hook 'cider-repl-mode-hook #'enable-paredit-mode))
```

inf-clojure `inf-clojure` is a third-party package offering basic integration with a running Clojure subprocess. This package is necessary for running a Figwheel process with Emacs. It's not as feature-rich as CIDER, but still offers the ability to load files, switch namespaces, evaluate expressions, show documentation, and do macro-expansion.

Currently disabled due to conflicts with `cider`

```
(use-package inf-clojure
  :ensure t
  :init
  (add-hook 'clojure-mode-hook #'inf-clojure-minor-mode))
```

Now lets write a simple function to run Figwheel as a Clojure subprocess.

```
(defun figwheel-repl ()
  (interactive)
  (inf-clojure "lein figwheel"))
```

Linting Clojure The flycheck-clojure package allows syntax checking for Clojure(Script). It uses eastwood, core.typed and kibit to lint Clojure(Script) through CIDER.

```
(use-package flycheck-clojure
  :ensure t
  :after cider flycheck
  :config
  (flycheck-clojure-setup))
```

Okay. There's been some snares getting this package to work, but with the help of this blogpost from the `flycheck-clojure` repo. (note to self: READMEs are friends), I'm beginning to make progress.

After cloning the project repo from my local file system, my debugging process has consisted of the following:

1. Navigate to the sample-project in the `squiggly-clojure` project repo.

2. Open `core.clj`
3. Launch an nrepl with Cider.
4. See `flycheck-clojure` being weird.
5. Annoyed Google search.
6. Edit my `clojure` configuration based on the last blog post.
7. Restart Emacs.
8. Repeat.

After running into problems documented in issues #45, #13, and #46, ~~I finally was able to get `flycheck-clojure` to stop doing weird things~~.

~~It's now doing nothing at all.~~

The project maintainers provide an example-config for setting up emacs, cider, flycheck, and friends. I messed around with this config for about half an hour and the latency issues and general inconsistency are the same.

I'm guessing I need to actually include the linters in my project's `project.clj`, but it's weird this package worked at all for bits and stretches if the dependencies need to be installed manually.

Typed Clojure Let's give this guy a try.

```
(use-package typed-clojure-mode
   :ensure t
   :after clojure-mode
   :init
   (add-hook 'clojure-mode-hook 'typed-clojure-mode))
```

```
java.lang.GoEFUrself!
```

```
# CompilerException java.lang.RuntimeException: Unable to resolve symbol: sym in thi
```

I've now stumbled upon `cider--debug-mode`.

This mode cannot be called manually, but with `C-u C-M-x` instead, and now `flycheck-clojure` ~~appears to be sort of working~~.

3 months later...

The above strikethrough denotes the point in time where I officially said FI.

Flycheck-Pos-Tip The `flycheck-clojure` repository recommendeds to install flycheck-pos-tip to keep linting and type errors from clashing with CIDER eldoc information.

```
(use-package flycheck-pos-tip
  :ensure t
  :after flycheck
  :init
  (flycheck-pos-tip-mode)
  :config
  (setq flycheck-display-errors-function
        #'flycheck-pos-tip-error-messages))
```

Java

Eclim

```
(use-package eclim
  :ensure t
  :if (eq system-type 'darwin)
  ;; load my forked version
  ;; :load-path "site-lisp/emacs-eclim/"
  :config
  (setq eclim-eclipse-dirs '("/Applications/Eclipse.app/Contents/Eclipse")
        eclim-executable "/Applications/Eclipse.app/Contents/Eclipse/eclim"
        eclimd-executable "/Applications/Eclipse.app/Contents/Eclipse/eclimd"
        eclimd-default-workspace "~/Developer/Projects/Java/Workspace"
        eclimd-autostart-with-default-workspace t
        eclim-autostart nil
        eclim-wait-for-process t))
```

LaTeX

Tex Config

```
(use-package tex
  :ensure auctex
  :defines latex-nofill-env
  :functions chaffin/tex-auto-fill-mode
  :init
  (progn
    (setq TeX-command-default "LaTeX"
          TeX-engine ’xetex
          TeX-auto-save t
          TeX-parse-self t
          TeX-syntactic-comment t
          TeX-source-correlate-start-server nil
          LaTeX-fill-break-at-separators nil)
    (defvar latex-nofill-env ’("equation"
                               "equation*"
                               "align"
                               "align*"
                               "tabular"
                               "tikzpicture"))
    (defun chaffin//tex-autofill ()
      "Check whether the pointer is currently inside one of
the environments in ‘latex-nofill-env‘ and inhibits auto-filling
of the current paragraph."
      (let ((do-auto-fill t)
            (current-environment "")
            (level 0))
        (while (and do-auto-fill (not (string- current-environment "document")))
          (setq level (1+ level)
                current-environment (LaTeX-current-environment level)
                do-auto-fill (not (member current-environment latex-nofill-env)))))
      (when do-auto-fill
        (do-auto-fill)))

    (defun chaffin/tex-auto-fill-mode ()
      (interactive)
      (auto-fill-mode)
      (setq auto-fill-mode ’chaffin/tex-autofill))

    (add-hook ’LaTeX-mode-hook ’chaffin/tex-auto-fill-mode)
    (add-hook ’LaTeX-mode-hook ’LaTeX-math-mode)
    (add-hook ’LaTeX-mode-hook ’TeX-PDF-mode)
    (add-hook ’LaTeX-mode-hook ’smartparens-mode)
    ;; Company AucTeX
      (use-package company-auctex
        :ensure t
        :init
        (company-auctex-init))
    ;; RefTeX
      (defun jchaffin/init-reftex ()
        (add-hook 'LaTeX-mode-hook 'turn-on-reftex)
        (setq reftex-plug-into-AUCTeX '(nil nil t t t)
              reftex-use-fonts t
              reftex-default-bibliography '("~/Dropbox/org/papers/references.bib")))
    (jchaffin/init-reftex)
    ;; Bibtex Config

    ;; Magic Latex Buffer
      (use-package magic-latex-buffer
        :ensure t
        :init
        (progn
          (add-hook 'LaTeX-mode-hook 'magic-latex-buffer)
          (setq magic-latex-enable-block-highlight t
                magic-latex-enable-suscript t
                magic-latex-enable-pretty-symbols t
                magic-latex-enable-block-align t
                magic-latex-enable-inline-image t)))
    ;; Latex Preview Pane
    ;; buggy
    ;;   (use-package latex-preview-pane
    ;;     :ensure t
    ;;     :config (latex-preview-pane-enable))

    ;; Org Edit Latex Buffer
      (use-package org-edit-latex
        :ensure t)
    ;; Latex Extra
    (use-package latex-extra
      :ensure t
      :init
      (add-hook 'LaTeX-mode-hook 'latex-extra-mode))
    ;; Auctex latexmk
    (use-package auctex-latexmk
      :ensure t
      :config
      (setq auctex-latexmk-inherit-TeX-PDF-mode t))
```

```
      ))
```

Bibtex #+NAME bibtex-config

```
(setq bibtex-autokey-year-length 4
      bibtex-autokey-name-year-separator "-"
      bibtex-autokey-year-title-separator "-"
      bibtex-autokey-titleword-separator "-"
      bibtex-autokey-titlewords 2
      bibtex-autokey-titlewords-stretch 1
      bibtex-autokey-titleword-length 5)
```

LaTeX Extra

```
(use-package latex-extra
  :ensure t
  :init
  (add-hook 'LaTeX-mode-hook 'latex-extra-mode))
```

LaTeX Preview Pane

```
  (use-package latex-preview-pane
    :ensure t
    :config (latex-preview-pane-enable))
```

Company AucTeX

```
  (use-package company-auctex
    :ensure t
    :init
    (company-auctex-init))
```

Org Edit Latex

```
  (use-package org-edit-latex
    :ensure t)
```

Magic Latex Buffer Prettify dedicated org-mode latex buffers.

```
(use-package magic-latex-buffer
  :ensure t
  :init
  (progn
    (add-hook 'LaTeX-mode-hook 'magic-latex-buffer)
    (setq magic-latex-enable-block-highlight t
          magic-latex-enable-suscript t
          magic-latex-enable-pretty-symbols t
          magic-latex-enable-block-align t
          magic-latex-enable-inline-image t)))
```

Auctex Latexmk

```
(use-package auctex-latexmk
  :ensure t
  :config
  (setq auctex-latexmk-inherit-TeX-PDF-mode t))
```

RefTeX RefTeX is a citation and reference tool maintained by the AucTeX team.

Since Emacs 24.3, its built in with the Emacs distribution.

```
(defun jchaffin/init-reftex ()
  (add-hook 'LaTeX-mode-hook 'turn-on-reftex)
  (setq reftex-plug-into-AUCTeX '(nil nil t t t)
        reftex-use-fonts t
        reftex-default-bibliography '("~/Dropbox/org/papers/references.bib")))
```

Texinfo #+texinfo-config

```
(use-package texinfo
  :ensure t
  :defines texinfo-section-list
  :commands texinfo-mode
  :init
  (add-to-list 'auto-mode-alist '("\\.texi$" . texinfo-mode)))
```

Javascript

JavaScript Preamble

```
(use-package js2-mode
  :ensure t
  :mode (("\\.js\\'" . js2-mode))
  :config
  (setq js-indent-level 2))
(use-package coffee-mode
  :ensure t
  :mode ("\\.coffee\\'" . coffee-mode))
(use-package json-mode
  :defer t
  :ensure t
  :mode (("\\.json\\'" . json-mode)))
(use-package tern
  :ensure t
  :after js2-mode
  :init (add-hook 'js2-mode-hook 'tern-mode))
```

js2-mode

```
(use-package js2-mode
  :ensure t
  :mode (("\\.js\\'" . js2-mode))
  :config
  (setq js-indent-level 2))
```

rjsx-mode Real jsx support.

```
(use-package rjsx-mode
  :ensure t
  :mode "\\.jsx\\'")
```

Tern Tern is a code-analysis engine for JavaScript.

```
(use-package tern
  :ensure t
  :after js2-mode
  :init (add-hook 'js2-mode-hook 'tern-mode))
```

Company Tern (repository) Tern backend using company.

```
(use-package company-tern
  :ensure t
  :init
  (add-to-list 'company-backends 'company-tern)
  :config
  (setq company-tern-property-marker nil
        company-tern-meta-as-single-line t))
```

JSON

```
(use-package json-mode
  :defer t
  :ensure t
  :mode (("\\.json\\'" . json-mode)))
```

Based off/shamelessly copied and pasted from Spacemacs React layer.

```
(progn
  (define-derived-mode react-mode web-mode "react")
  (add-to-list 'auto-mode-alist '("\\.jsx\\'" . react-mode))
  (add-to-list 'auto-mode-alist '("\\.react.js\\'" . react-mode))
  (add-to-list 'auto-mode-alist '("\\.index.android.js\\'" . react-mode))
  (add-to-list 'auto-mode-alist '("\\.index.ios.js\\'" . react-mode))
  (add-to-list 'auto-mode-alist '("\\/\\*\\* @jsx .*\\*/\\'" . react-mode)))
```

Coffee

```
(use-package coffee-mode
  :ensure t
  :mode ("\\.coffee\\'" . coffee-mode))
```

Add Node Modules Path (repository)

```
(use-package add-node-modules-path
  :ensure t
  :defer t
  :init
  (add-hook 'js2-mode-hook #'add-node-modules-path))
```

Npm Mode

```
(use-package npm-mode
  :ensure t
  :defer t)
```

Enable syntax checking in `js-mode` and related mode buffers.

```
(dolist (mode '(coffee-mode js2-mode json-mode))
  (push mode flycheck-global-modes))
```

Web

Web Mode

```
(use-package web-mode
  :ensure t
  :bind (:map web-mode-map
              ("M-n" . web-mode-tag-match))
  :mode
  (("\\.phtml\\'"      . web-mode)
   ("\\.tpl\\.php\\'"  . web-mode)
   ("\\.twig\\'"       . web-mode)
   ("\\.html\\'"       . web-mode)
   ("\\.htm\\'"        . web-mode)
   ("\\.[gj]sp\\'"     . web-mode)
   ("\\.as[cp]x?\\'"   . web-mode)
   ("\\.eex\\'"        . web-mode)
   ("\\.erb\\'"        . web-mode)
   ("\\.mustache\\'"   . web-mode)
   ("\\.handlebars\\'" . web-mode)
   ("\\.hbs\\'"        . web-mode)
   ("\\.eco\\'"        . web-mode)
   ("\\.ejs\\'"        . web-mode)
   ("\\.djhtml\\'"     . web-mode))

  :config
  (progn
    (setq web-mode-engines-alist
        '(("php" . "\\.phtml\\'")
          ("blade" . "\\.blade\\'")))
```

```
(defun jchaffin/web-mode-enable ()
  (setq web-mode-enable-auto-pairing t
        web-mode-enable-css-colorization t
        web-mode-enable-block-face t
        web-mode-enable-part-face t
        web-mode-enable-comment-keywords t
        web-mode-enable-heredoc-fontification t
        web-mode-enable-current-element-highlight t
        web-mode-enable-current-column-highlight t))

(add-hook 'web-mode-hook #'jchaffin/web-mode-enable)


(defun jchaffin/web-mode-indent ()
  (setq web-mode-markup-indent-offset 2
        web-mode-code-indent-offset 2
        web-mode-style-padding 1
        web-mode-script-padding 1
        web-mode-block-padding 0
        web-mode-comment-style 2))

(add-hook 'web-mode-hook #'jchaffin/web-mode-indent)

  (use-package tagedit
    :ensure t
    :diminish tagedit-mode
    :config
    (progn
      (tagedit-add-experimental-features)
      (add-hook 'html-mode-hook (lambda () (tagedit-mode 1)))))

  (use-package emmet-mode
    :ensure t
    :defer t
    :init
    (add-to-hooks 'emmet-mode '(css-mode-hook
                                html-mode-hook
                                web-mode-hook)))
  (use-package company-web
          :ensure t
          :init
          (progn
              (use-package company-tern
                :ensure t
                :init
                (add-to-list 'company-backends 'company-tern)
                :config
                (setq company-tern-property-marker nil
                      company-tern-meta-as-single-line t))
            (defun jchaffin/company-web-mode-hook ()
              "Autocompletion hook for web-mode"
              (set (make-local-variable 'company-backends)
                   '(company-tern company-web-html company-yasnippet company-files)))

            (add-hook 'web-mode-hook #'jchaffin/company-web-mode-hook)

            ;; Enable JavaScript completion between <script>...</script> etc.
            (defadvice company-tern (before web-mode-set-up-ac-sources activate)
              "Set `tern-mode' based on current language before running company-tern."
              (if (equal major-mode 'web-mode)
                  (let ((web-mode-cur-language
                         (web-mode-language-at-pos)))
                    (if (or (string= web-mode-cur-language "javascript")
                            (string= web-mode-cur-language "jsx"))
                        (unless tern-mode (tern-mode))
                      (if tern-mode (tern-mode -1))))))))))
```

Emmet

```
(use-package emmet-mode
  :ensure t
  :defer t
  :init
  (add-to-hooks 'emmet-mode '(css-mode-hook
                              html-mode-hook
                              web-mode-hook)))
```

HTML

Tag Edit

```
(use-package tagedit
  :ensure t
  :diminish tagedit-mode
  :config
  (progn
    (tagedit-add-experimental-features)
    (add-hook 'html-mode-hook (lambda () (tagedit-mode 1)))))
```

CSS

CSS Mode

```
(use-package css-mode
  :ensure t
  :defer t
  :commands css-expand-statment css-contract-statement
  :bind (("C-c c z" . css-contract-statement)
         ("C-c c o" . css-expand-statement))
  :init
  (progn
    (defun css-expand-statment ()
      (interactive)
      (save-excursion
        (end-of-line)
        (search-backward "{")
        (forward-char 1)
        (while (or (eobp) (not (looking-at "}")))
          (let ((beg (point)))
            (newline)
            (search-forward ";")
            (indent-region beg (point))))
        (newline)))

    (defun css-contrac-statement ()
      "Contract CSS Block"
      (interactive)
      (end-of-line)
      (search-backward "{")
      (while (not (looking-at "}"))
        (join-line -1)))))
```

Less

```
(use-package css-less-mode
  :ensure t
  :mode ("\\.less\\'" . less-css-mode))
```

SASS

```
(use-package sass-mode
  :ensure t
  :mode ("\\.sass\\'" . sass-mode))
```

SCSS

```
(use-package scss-mode
  :ensure t
  :mode ("\\.scss\\'" . scss-mode))
```

Syntax Checking and Code Completion

Company Web (repository) Code completion for html-mode, web-mode, jade-mode, and slim-mode using company.

```
(use-package company-web
  :ensure t
  :init
  (progn
      (use-package company-tern
        :ensure t
        :init
        (add-to-list 'company-backends 'company-tern)
        :config
        (setq company-tern-property-marker nil
              company-tern-meta-as-single-line t))
    (defun jchaffin/company-web-mode-hook ()
      "Autocompletion hook for web-mode"
      (set (make-local-variable 'company-backends)
           '(company-tern company-web-html company-yasnippet company-files)))

    (add-hook 'web-mode-hook #'jchaffin/company-web-mode-hook)

    ;; Enable JavaScript completion between <script>...</script> etc.
    (defadvice company-tern (before web-mode-set-up-ac-sources activate)
      "Set `tern-mode' based on current language before running company-tern."
      (if (equal major-mode 'web-mode)
          (let ((web-mode-cur-language
                 (web-mode-language-at-pos)))
            (if (or (string= web-mode-cur-language "javascript")
                    (string= web-mode-cur-language "jsx"))
                (unless tern-mode (tern-mode))
              (if tern-mode (tern-mode -1))))))))
```

Markdown

Markdown Mode

- See https://jblevins.org/projects/markdown-mode/

  #+NAME markdown-mode-config

```
(use-package markdown-mode
  :ensure t
  :commands (markdown-mode gfm-mode)
  :mode (("README\\.md\\'" . gfm-mode)
         ("\\.md\\'" . markdown-mode)
         ("\\.markdown\\'" . markdown-mode))
  :init
  (progn
    (setq markdown-command "multimarkdown")
    (when *is-mac*
      ;; FIX ME
      (setq markdown-open-command (lambda () (shell-command "open -a Marked" buffer-
```

markdown-mode+

```
(use-package markdown-mode+
  :if *is-mac*
  :ensure t)
```

Ruby

ruby-mode

```
(use-package ruby-mode
  :mode "\\.rb\\'"
  :interpreter "ruby"
  :functions inf-ruby-keys
  :config
  (defun chaffin/ruby-mode-hook ()
    (require 'inf-ruby)
    (inf-ruby-keys))

  (add-hook #'ruby-mode-hook #'chaffin/ruby-mode-hook))
```

Docker

```
(use-package dockerfile-mode
  :ensure t
  :mode ( "Dockerfile\\'" .  dockerfile-mode))
```