The Heretic's Emacs config

Jack Henahan

January 18, 2018

Contents

1 Basics

1.1 Modern Libraries

1.1.1 Libraries

1. Async The `async` library makes writing asynchronous functions easier (since async in Emacs is kind of a nightmare, otherwise).

```
(use-package async
  :commands (async-start))
```

2. `cl-lib` This library extends the "Lispiness" of Elisp, making it more like Common Lisp though I always feel a bit dirty about using it.

```
(use-package cl-lib
  :demand t)
```

3. Lists The list API in Elisp basically sucks, so there's `dash`. I'm attempting to learn how to use `seq` properly, though, since it's built in and should be just about at parity with `dash`'s API, if a bit more verbose.

```
(use-package dash
  :demand t)
```

4. Files The file API in Elisp **also** sucks, so we have `f.el`.

   ```
   (use-package f
     :demand t)
   ```

5. Strings Keeping up the pattern, the string API in Elisp is hot garbage, so we have `s.el` to make things nice.

   ```
   (use-package s
     :demand t)
   ```

6. Hash Tables Elisp's hash table API is based around `make-hash-table`, which is a pretty awful function with five keyword arguments. This might almost be worth it for how much faster hash tables are than alists, but luckily `ht.el` means we don't have to choose.

   ```
   (use-package ht
     :demand t)
   ```

7. `subr` extensions `subr-x` provides some built-in equivalents to some of the modern APIs above, some implemented in C.

   ```
   (require 'subr-x)
   ```

1.2 Sensible Defaults

1.2.1 Default Directory

When using `find-file`, search from the user's home directory.

```
(setq default-directory "~/")
```

1.2.2 Treat Camel-Case Words as separate words

```
(add-hook 'prog-mode-hook 'subword-mode)
```

1.2.3 Increase GC threshold

Allow 20MB of memory (instead of 0.76MB) before calling garbage collection. This means GC runs less often, which speeds up some operations.

```
(setq gc-cons-threshold 20000000)
```

1.2.4 Make Scripts Executable By Default

If your file starts with a shebang, the file will be marked executable on save.

```
(add-hook 'after-save-hook
          'executable-make-buffer-file-executable-if-script-p)
```

1.2.5 Transient Mark Mode

Transient mark means region highlighting works the way you would expect it to coming from other editors.

```
(transient-mark-mode t)
```

1.2.6 Short Confirmations

Typing out `yes` and `no` is irritating. Just use `y` or `n`.

```
(fset #'yes-or-no-p #'y-or-n-p)
```

1.2.7 macOS settings

If you set Emacs as the default file handler for certain types of files, double-clicking will open an entire new Emacs frame. This setting causes Emacs to reuse the existing one.

```
(the-with-operating-system macOS
  (setq ns-pop-up-frames nil))
```

1.3 Enable Disabled Commands

It is obvious to anyone that if a function is disabled then it must be powerful, or at least interesting. I want them.

1.3.1 Do it

```
(setq disabled-command-function nil)
```

1.4 Killing and Yanking (copying and pasting)

1.4.1 Settings

1. Delete Selection If you start typing when you have something selected, then the selection will be deleted. If you press DEL while you have something selected, it will be deleted rather than killed. (Otherwise, in both cases the selection is deselected and the normal function of the key is performed.)

   ```
   (delete-selection-mode 1)
   ```

   (a) AUCTeX compatibility Make delete-selection-mode work properly with AUCTeX.

   ```
   (with-eval-after-load 'latex
     (put 'LaTeX-insert-left-brace 'delete-selection t))
   ```

2. Eliminate duplicates in kill ring If you kill the same thing twice, you won't have to use `yank` twice to get past it to older entries in the kill ring.

   ```
   (setq kill-do-not-save-duplicates t)
   ```

1.5 Startup

By default, Emacs fills your universe with (FSF-approved and GPL-compliant) garbage. I don't want a bit of it.

1.5.1 Disable Startup Message

I like GNU. You maybe like GNU. You're using Emacs. Whatever. You don't need the "For information about Emacs..." message.

```
(defalias 'the--advice-inhibit-startup-echo-area-message #'ignore
  "Unconditionally inhibit the startup message in the echo area.
This is an ‘:override’ advice for
‘display-startup-echo-area-message’.")

(advice-add #'display-startup-echo-area-message :override
            #'the--advice-inhibit-startup-echo-area-message)
```

1.5.2 Disable About

If I wanted to know about GNU Emacs, there are dozens of doc options. I do not need that buffer on startup. We disable that and instead use our own fancy dashboard.

```
(use-package dashboard
  :demand t
  :after (org-agenda projectile)
  :config
  (setq recentf-exclude (-map 'f-canonical (org-agenda-files)))
  (setq dashboard-banner-logo-title "REPENT!")
  (setq dashboard-startup-banner (f-expand "heresy.png" the-image-directory))
  (setq dashboard-items '((recents  . 5)
                          (bookmarks . 5)
                          (projects . 5)
                          (agenda . 5)
                          (registers . 5)))
  (dashboard-setup-startup-hook))
```

1.5.3 Blank Scratch Buffer

I know what a scratch buffer is. Hush.

```
(setq initial-scratch-message nil)
```

1.5.4 Emacs Server

Start up an Emacs server process so we can attach `emacsclient` to it and get that fast response time the Vim people are so smug about.

```
(server-start)
```

1.6 Auto-Revert

1.6.1 Settings

Turn the delay on auto-reloading from 5 seconds down to 1 second. We have to do this before turning on `auto-revert-mode` for the change to take effect, unless we do it through `customize-set-variable` (which is slow enough to show up in startup profiling).

```
(setq auto-revert-interval 1)
```

Automatically reload files that were changed on disk, if they have not been modified in Emacs since the last time they were saved.

```
(global-auto-revert-mode 1)
```

Only automatically revert buffers that are visible. This should improve performance (because if you have 200 buffers open...). This code is originally based on this Emacs.SE thread.

Note that calling `global-auto-revert-mode` above triggers an autoload for `autorevert`, so there's no need to 'require' it again here.

```
(el-patch-defun auto-revert-buffers ()
  "Revert buffers as specified by Auto-Revert and Global Auto-Revert Mode.
Should ‘global-auto-revert-mode’ be active all file buffers are checked.
Should ‘auto-revert-mode’ be active in some buffers, those buffers
are checked.
Non-file buffers that have a custom ‘revert-buffer-function’ and
‘buffer-stale-function’ are reverted either when Auto-Revert
Mode is active in that buffer, or when the variable
‘global-auto-revert-non-file-buffers’ is non-nil and Global
Auto-Revert Mode is active.
This function stops whenever there is user input.  The buffers not
checked are stored in the variable ‘auto-revert-remaining-buffers’.
To avoid starvation, the buffers in ‘auto-revert-remaining-buffers’
are checked first the next time this function is called.
This function is also responsible for removing buffers no longer in
Auto-Revert mode from ‘auto-revert-buffer-list’, and for canceling
the timer when no buffers need to be checked."

  (setq auto-revert-buffers-counter
        (1+ auto-revert-buffers-counter))

  (save-match-data
    (let ((bufs (el-patch-wrap 2
                  (cl-remove-if-not
                   #'get-buffer-window
                   (if global-auto-revert-mode
                       (buffer-list)
                     auto-revert-buffer-list))))
          remaining new)
```

```
;; Partition ‘bufs’ into two halves depending on whether or not
;; the buffers are in ‘auto-revert-remaining-buffers’.  The two
;; halves are then re-joined with the "remaining" buffers at the
;; head of the list.
(dolist (buf auto-revert-remaining-buffers)
  (if (memq buf bufs)
      (push buf remaining)))
(dolist (buf bufs)
  (if (not (memq buf remaining))
      (push buf new)))
(setq bufs (nreverse (nconc new remaining)))
(while (and bufs
            (not (and auto-revert-stop-on-user-input
                      (input-pending-p))))
  (let ((buf (car bufs)))
    (if (buffer-live-p buf)
        (with-current-buffer buf
          ;; Test if someone has turned off Auto-Revert Mode in a
          ;; non-standard way, for example by changing major mode.
          (if (and (not auto-revert-mode)
                   (not auto-revert-tail-mode)
                   (memq buf auto-revert-buffer-list))
              (setq auto-revert-buffer-list
                    (delq buf auto-revert-buffer-list)))
          (when (auto-revert-active-p)
            ;; Enable file notification.
            (when (and auto-revert-use-notify
                       (not auto-revert-notify-watch-descriptor))
              (auto-revert-notify-add-watch))
            (auto-revert-handler)))
      ;; Remove dead buffer from ‘auto-revert-buffer-list’.
      (setq auto-revert-buffer-list
            (delq buf auto-revert-buffer-list))))
  (setq bufs (cdr bufs)))
(setq auto-revert-remaining-buffers bufs)
;; Check if we should cancel the timer.
(when (and (not global-auto-revert-mode)
           (null auto-revert-buffer-list))
  (cancel-timer auto-revert-timer)
  (setq auto-revert-timer nil)))))
```

Auto-revert all buffers, not only file-visiting buffers. The docstring warns about potential performance problems but this should not be an issue since we only revert visible buffers.

```
(setq global-auto-revert-non-file-buffers t)
```

Since we automatically revert all visible buffers after one second, there's no point in asking the user whether or not they want to do it when they find a file. This disables that prompt.

```
(setq revert-without-query '(".*"))
```

Prevent **Help** buffers from asking for confirmation about reverting.

```
(with-eval-after-load 'help-mode
  (defun the--advice-disable-help-mode-revert-prompt
      (help-mode-revert-buffer _ignore-auto _noconfirm)
    (funcall help-mode-revert-buffer _ignore-auto 'noconfirm))
  (advice-add #'help-mode-revert-buffer :around
              #'the--advice-disable-help-mode-revert-prompt))
```

Don't show it in the mode line.

```
(setq auto-revert-mode-text nil)
```

1.7 Saving Files

2 Package Management

2.1 Package Management

2.1.1 Disable `package.el`

We use `straight.el` in this household, and like it! Emacs will initialize `package.el` unless we tell it not to, so we do so.

```
(setq package-enable-at-startup nil)
```

2.1.2 Bootstrap `straight.el`

We are using a package manager called straight.el. This code, which is taken from the README, bootstraps the system (because obviously the package manager is unable to install and load itself, if it is not already installed and loaded).

```
(let ((bootstrap-file (concat user-emacs-directory "straight/repos/straight.el/bootstra
      (bootstrap-version 3))
  (unless (file-exists-p bootstrap-file)
    (with-current-buffer
        (url-retrieve-synchronously
         "https://raw.githubusercontent.com/raxod502/straight.el/develop/install.el"
         'silent 'inhibit-cookies)
      (goto-char (point-max))
      (eval-print-last-sexp)))
  (load bootstrap-file nil 'nomessage))
```

2.1.3 `use-package`

To handle a lot of useful tasks related to package configuration, we use a library called 'use-package', which provides a macro by the same name. This macro automates many common tasks, like autoloading functions, binding keys, registering major modes, and lazy-loading, through the use of keyword arguments. See the README.

```
(straight-use-package 'use-package)
```

1. `straight.el` integration Tell use-package to automatically install packages if they are missing. By default, packages are installed via `straight.el`, which draws package installation recipes (short lists explaining where to download the package) from MELPA, GNU ELPA, and EmacsMirror. (But you can also specify a recipe manually by putting `:straight` in the `use-package` call, which is an extension to `use-package` provided by `straight.el`.) Learn more about recipe formatting from the MELPA README.

   ```
   (setq straight-use-package-by-default t)
   ```

2. Lazy-loading Tell use-package to always load packages lazily unless told otherwise. It's nicer to have this kind of thing be deterministic:

if ‘:demand’ is present, the loading is eager; otherwise, the loading is lazy. See the `use-package` documentation.

```
(setq use-package-always-defer t)
```

2.2 Future-proof patches

3 Customization

3.1 Customization Group

3.1.1 THE group

Here we define a customization group for THE. This allows users to customize the variables declared here in a user-friendly way using the Custom interface.

```
(defgroup the nil
  "Customize your THE experience"
  :group 'emacs)
```

4 System Integration

4.1 OS Integration

4.1.1 OS-specific Customization

1. OS detection Ideally, we detect the operating system here, as well as giving the option to customize it directly in case we get it wrong.

```
(defcustom the-operating-system
  (pcase system-type
    ('darwin 'macOS)
    ((or 'ms-dos 'windows-nt 'cygwin) 'windows)
    (_ 'linux))
  "Specifies the operating system.
This can be ‘macOS’, ‘linux’, or ‘windows’. Normally this is
automatically detected and does not need to be changed."
  :group 'the
  :type '(choice (const :tag "macOS" macOS)
                 (const :tag "Windows" windows)
                 (const :tag "Linux" linux)))
```

2. OS-specific settings This macro allows us to handle things like operating system specific clipboard/mouse hacks and other things like that.

```
(defmacro the-with-operating-system (os &rest body)
  "If the operating system is OS, eval BODY.
See ‘the-operating-system’ for the possible values of OS,
which should not be quoted."
  (declare (indent 1))
  ‘(when (eq the-operating-system ’,os)
     ,@body))
```

4.1.2 Path Settings

1. Path Fixes In the terminal, the mouse and clipboard don't work properly. But in windowed Emacsen, the `PATH` is not necessarily set correctly! You can't win.

```
(use-package exec-path-from-shell
  :demand t
  :config
  (the-with-operating-system macOS
    (exec-path-from-shell-initialize))
  (the-with-operating-system linux
    (exec-path-from-shell-initialize)))
```

4.1.3 Clipboard Integration

1. macOS integration Like mouse integration, clipboard integration works properly in windowed Emacs but not in terminal Emacs (at least by default). This code was originally based on 1, and then modified based on 2.

```
(the-with-operating-system macOS
  (the-with-terminal-emacs
    (defvar the-clipboard-last-copy nil
      "The last text that was copied to the system clipboard.
This is used to prevent duplicate entries in the kill ring.")

    (defun the-clipboard-paste ()
      "Return the contents of the macOS clipboard, as a string."
```

```
  (let* (;; Setting ‘default-directory’ to a directory that is
         ;; sure to exist means that this code won’t error out
         ;; when the directory for the current buffer does not
         ;; exist.
         (default-directory "/")
         ;; Command pbpaste returns the clipboard contents as a
         ;; string.
         (text (shell-command-to-string "pbpaste")))
    ;; If this function returns nil then the system clipboard is
    ;; ignored and the first element in the kill ring (which, if
    ;; the system clipboard has not been modified since the last
    ;; kill, will be the same). Including this ‘unless’ clause
    ;; prevents you from getting the same text yanked the first
    ;; time you run ‘yank-pop’. (Of course, this is less relevant
    ;; due to ‘counsel-yank-pop’, but still arguably the correct
    ;; behavior.)
    (unless (string= text the-clipboard-last-copy)
      text)))

(defun the-clipboard-copy (text)
  "Set the contents of the macOS clipboard to given TEXT string."
  (let* (;; Setting ‘default-directory’ to a directory that is
         ;; sure to exist means that this code won’t error out
         ;; when the directory for the current buffer does not
         ;; exist.
         (default-directory "/")
         ;; Setting ‘process-connection-type’ makes Emacs use a pipe to
         ;; communicate with pbcopy, rather than a pty (which is
         ;; overkill).
         (process-connection-type nil)
         ;; The nil argument tells Emacs to discard stdout and
         ;; stderr. Note, we aren’t using ‘call-process’ here
         ;; because we want this command to be asynchronous.
         ;;
         ;; Command pbcopy writes stdin to the clipboard until it
         ;; receives EOF.
         (proc (start-process "pbcopy" nil "pbcopy")))
    (process-send-string proc text)
    (process-send-eof proc))
  (setq the-clipboard-last-copy text))
```

```
(setq interprogram-paste-function #'the-clipboard-paste)
(setq interprogram-cut-function #'the-clipboard-copy)))
```

2. Inter-program paste If you have something on the system clipboard, and then kill something in Emacs, then by default whatever you had on the system clipboard is gone and there is no way to get it back. Setting the following option makes it so that when you kill something in Emacs, whatever was previously on the system clipboard is pushed into the kill ring. This way, you can paste it with `yank-pop`.

```
(setq save-interprogram-paste-before-kill t)
```

4.2 UI Integration

4.2.1 Window System

These macros give us a convenient way to conditionally execute code based on the active window system.

1. Macros

```
(defmacro the-with-windowed-emacs (&rest body)
  "Eval BODY if Emacs is windowed, else return nil."
  (declare (indent defun))
  `(when (display-graphic-p)
     ,@body))

(defmacro the-with-terminal-emacs (&rest body)
  "Eval BODY if Emacs is not windowed, else return nil."
  (declare (indent defun))
  `(unless (display-graphic-p)
     ,@body))
```

5 Config Management

5.1 Config File Utilities

5.1.1 Apache configs

```
(use-package apache-mode)
```

5.1.2 Dockerfiles

```
(use-package dockerfile-mode)
```

5.1.3 Git files

1. Git config and modules

   ```
   (use-package gitconfig-mode)
   ```

2. Git ignore files

   ```
   (use-package gitignore-mode)
   ```

5.1.4 SSH configs

```
(use-package ssh-config-mode)
```

5.1.5 YAML

I edit a lot of YAML files, especially Ansible configs. The current version of `emacs-ansible` has a hard dependency on `auto-complete`, which we don't use, so until there's a version without that dependency, we just turn `yaml-mode` on whenever we think we're in an Ansible file. `auto-fill` is also turned off in YAML buffers because it breaks things.

```
(use-package yaml-mode
  :init
  (setq the--ansible-filename-re ".*\\(main\.yml\\|site\.yml\\|encrypted\.yml\\|roles/
  (add-to-list 'auto-mode-alist `(,the--ansible-filename-re . yaml-mode))
  :config
  (defun the--disable-auto-fill-mode ()
    (auto-fill-mode -1))

  (add-hook 'yaml-mode-hook #'the--disable-auto-fill-mode))
```

5.1.6 Jinja2

Jinja2 is the template format of record for Ansible, so we just add basic support here.

```
(use-package jinja2-mode)
```

5.2 Emacs

5.2.1 emacs.d Organization

1. `no-littering` A lot of packages (and also a lot of Emacs defaults) throw files all over your config directory. `no-littering` sets a lot of sensible defaults for commonly used packages to keep the config directory manageable and discoverable.

```
(use-package no-littering
  :demand t)
```

6 Documentation

6.1 Better Help

6.1.1 Helpful

We alias common help methods to their Helpful equivalents because Helpful is a much nicer version of the built-in help.

```
(use-package helpful
  :demand t
  :config
  (defalias #'describe-key #'helpful-key)
  (defalias #'describe-function #'helpful-callable)
  (defalias #'describe-variable #'helpful-variable)
  (defalias #'describe-symbol #'helpful-symbol))
```

6.2 ElDoc

6.2.1 Settings

Show ElDoc messages in the echo area immediately, instead of after 1/2 a second.

```
(setq eldoc-idle-delay 0)
```

Always truncate ElDoc messages to one line. This prevents the echo area from resizing itself unexpectedly when point is on a variable with a multiline docstring.

```
(setq eldoc-echo-area-use-multiline-p nil)
```

Don't show ElDoc in the mode line.

```
(setq eldoc-minor-mode-string nil)
```

Slow down ElDoc if metadata fetching is causing performance issues.

```
(defun the-eldoc-toggle-slow ()
  "Slow down ‘eldoc’ by turning up the delay before metadata is shown.
This is done in ‘the-slow-autocomplete-mode’."
  (if the-slow-autocomplete-mode
      (setq-local eldoc-idle-delay 1)
    (kill-local-variable 'eldoc-idle-delay)))

(add-hook 'the-slow-autocomplete-mode-hook #'the-eldoc-toggle-slow)
```

7 Keybindings

7.1 Binding Keys

7.1.1 Custom Prefix

There's a lot of room for keybindings, but we rely on a common prefix for discoverability and to leave room for extension. This also makes creating modal bindings later quite a bit easier.

```
(defcustom the-prefix "M-T"
  "Prefix key sequence for The-related keybindings.
This is a string as would be passed to ‘kbd’."
  :group 'the
  :type 'string)
```

For convenience, we also have a function that will create binding strings using our prefix. This mainly gets used in bind-key declarations until I can figure out how to evaluate code in org-table cells to make the whole thing more customizable.

```
(defun the-join-keys (&rest keys)
  "Join key sequences. Empty strings and nils are discarded.
\(the--join-keys \"M-P e\" \"e i\") => \"M-P e e i\"
\(the--join-keys \"M-P\" \"\" \"e i\") => \"M-P e i\""
  (string-join (remove "" (mapcar #'string-trim (remove nil keys))) " "))
```

7.1.2 `bind-key`

`bind-key` is the prettier cousin of `define-key` and `global-set-key`, as well as providing the `:bind` family of keywords in `use-package`,

```
(use-package bind-key)
```

7.2 Hydra

Hydras are a really fancy feature that let you create families of related bindings with a common prefix.

7.2.1 `use-package` declaration

```
(use-package hydra
  :demand t)
```

8 UI

8.1 Appearance

8.1.1 Basic Setup

This file has appearance tweaks that are unrelated to the color theme. Menus, scroll bars, bells, cursors, and so on. See also `the-theme`, which customizes the color theme specifically.

8.1.2 Fullscreen

I use `chunkwm` to manage most windows, including Emacs, so the native fullscreen mode is unnecessary. It's also necessary to set pixelwise frame resizing non-nil for a variety of window managers. I don't see any particular harm in having it on, regardless of WM.

```
(the-with-operating-system macOS
  (setq ns-use-native-fullscreen nil))
(setq frame-resize-pixelwise t)
```

8.1.3 Interface Cleanup

Emacs defaults are a nightmare of toolbars and scrollbars and such nonsense. We'll turn all of that off.

```
(menu-bar-mode -1)
(setq ring-bell-function #'ignore)
(scroll-bar-mode -1)
(tool-bar-mode -1)
(blink-cursor-mode -1)
```

8.1.4 Keystroke Display

Display keystrokes in the echo area immediately, not after one second. We can't set the delay to zero because somebody thought it would be a good idea to have that value suppress keystroke display entirely.

```
(setq echo-keystrokes 1e-6)
```

8.1.5 No Title Bars

I put a lot of effort into purging title bars from most of the software I use on a regular basis (what a waste of real estate), and in Emacs 26 (might really be 26.2 or so) this is built in. For earlier versions, patches exist to get the same effect.

```
(if (version<= "26" emacs-version)
    (setq default-frame-alist '((undecorated . t))))
```

8.1.6 Fonts

I use Pragmata Pro everywhere, but I'll eventually figure out how to deal with fonts properly and allow this to be specified.

8.1.7 Adjust font size by screen resolution

The biggest issue I have with multiple monitors is that the font size is all over the place. The functions below just set up some reasonable defaults and machinery to change the size depending on the resolution of the monitor.

```
(defun the-fontify-frame (frame)
  (interactive)
  (the-with-windowed-emacs
    (if (> (x-display-pixel-width) 2000)
        (set-frame-parameter frame 'font "PragmataPro 22") ;; Cinema Display
      (set-frame-parameter frame 'font "PragmataPro 16"))))
```

```
(defun the-fontify-this-frame ()
  (interactive)
  (the-fontify-frame nil))

(defun the-fontify-idle ()
  (interactive)
  (the-fontify-this-frame)
  (run-with-idle-timer 1 t 'the-fontify-this-frame))

(call-interactively 'the-fontify-idle)
```

8.2 Theme

8.2.1 Utilities

This function is useful for reformatting lispy names (like `deeper-blue`) into user-friendly strings (like "Deeper Blue") for the Custom interface.

```
(defun the--unlispify (name)
  "Converts \"deep-blue\" to \"Deep Blue\"."
  (capitalize
   (replace-regexp-in-string
    "-" " " name)))
```

This is a handy macro for conditionally enabling color theme customizations.

```
(defmacro the-with-color-theme (theme &rest body)
  "If the current color theme is THEME, eval BODY; else return nil.
The current color theme is determined by consulting
‘the-color-theme’."
  (declare (indent 1))
  ;; ‘theme’ should be a symbol so we can use ‘eq’.
  ‘(when (eq ',theme the-color-theme)
     ,@body))
```

8.2.2 Default Color Scheme

The default color scheme is Gruvbox, but you can set it to whatever you like.

```
(defcustom the-color-theme 'gruvbox
```

```
  "Specifies the color theme used by The.
You can use anything listed by ‘custom-available-themes’. If you
wish to use your own color theme, you can set this to nil."
  :group 'the
  :type ‘(choice ,@(mapcar (lambda (theme)
                             ‘(const :tag
                                     ,(if theme
                                          (the--unlispify
                                           (symbol-name theme))
                                        "None")
                                     ,theme))
                           (cons
                            nil
                            (sort
                             (append
                              (custom-available-themes)
                              '(leuven
                                gruvbox))
                             #'string-lessp)))))
```

Defer color theme loading until after init, which helps to avoid weirdness during the processing of the local init-file.

```
(defcustom the-defer-color-theme t
  "Non-nil means defer loading the color theme until after init.
Otherwise, the color theme is loaded whenever ‘the-theme’ is
loaded."
  :group 'the
  :type 'boolean)
```

8.2.3 Leuven Customization

- Change the highlight color from yellow to blue
- Don't underline current search match
- Lighten the search highlight face and remove the underline
- Don't underline mismatched parens

```
(the-with-color-theme leuven
```

```
    (set-face-background 'highlight "#B1EAFC")
    (set-face-underline 'isearch nil)
    (set-face-background 'lazy-highlight "#B1EAFC")
    (set-face-underline 'lazy-highlight nil)
    (set-face-underline 'show-paren-mismatch nil))
```

8.2.4 Gruvbox installation

We register the Gruvbox package with Straight, but it is only downloaded if the theme is active.

```
(straight-register-package 'gruvbox-theme)
(the-with-color-theme gruvbox
  (use-package gruvbox-theme))
```

8.2.5 Actually load the theme

Load the appropriate color scheme as specified in `the-color-theme`.

```
(when the-color-theme
  (if the-defer-color-theme
      (progn
        (eval-and-compile
          (defun the-load-color-theme ()
            "Load the The color theme, as given by `the-color-theme'.
If there is an error, report it as a warning."
            (condition-case-unless-debug error-data
                (load-theme the-color-theme 'no-confirm)
              (error (warn "Could not load color theme: %s"
                           (error-message-string error-data))))))
        (add-hook 'after-init-hook #'the-load-color-theme))
    (load-theme the-color-theme 'no-confirm)))
```

8.3 Modeline Configuration

8.3.1 Diminish

`diminish` allows us to change the display of minor modes in the modeline. I prefer Delight, but `diminish` is the standard used by many packages.

```
(use-package diminish
  :demand t)
```

8.3.2 Delight

`delight` allows us to change the display of minor and major modes in the modeline. Spaceline is gonna do a lot of this work for us, but for anything it doesn't catch we'll make our own lighter. This also gives us the `:delight` keyword in our `use-package` declarations.

```
(use-package delight
  :demand t
  :delight
  (abbrev-mode)
  (auto-fill-function)
  (eldoc-mode "ε")
  (emacs-lisp-mode "Ξ")
  (filladapt-mode)
  (outline-minor-mode)
  (smerge-mode)
  (subword-mode)
  (undo-tree-mode)
  (visual-line-mode "ω")
  (which-key-mode)
  (whitespace-mode)
  )
```

8.3.3 Nyan!

Gotta have that cat! This will add a Nyan Cat progress indicator to the modeline.

```
(use-package nyan-mode
  :demand t
  :init
  (setq nyan-animate-nyancat t
        nyan-wavy-trail t)
  :config
  (nyan-mode 1))
```

8.3.4 Spaceline - All-the-Icons

We use Spaceline for our modeline, along with a theme called `spaceline-all-the-icons` which uses rich icon fonts where appropriate.

1. Setup

   (a) Display diminished minor modes

   ```
   (spaceline-toggle-all-the-icons-minor-modes-on)
   ```

   (b) Change modeline color based on Modalka stat

   ```
   (defface the-spaceline-modalka-off
     '((t (:background "chartreuse3"
           :foreground "#3E3D31")))
     "Modalka inactive face."
     :group 'the)

   (defface the-spaceline-modalka-on
     '((t (:background "DarkGoldenrod2"
           :foreground "#3E3D31")))
     "Modalka inactive face."
     :group 'the)

   (defun the-spaceline-modalka-highlight ()
     (if modalka-mode
         'the-spaceline-modalka-on
       'the-spaceline-modalka-off))

   (setq spaceline-highlight-face-func #'the-spaceline-modalka-highlight)
   ```

   (c) Turn on Nyan Cat

   ```
   (spaceline-toggle-all-the-icons-nyan-cat-on)
   ```

2. `use-package` declaration

```
(use-package spaceline-all-the-icons
  :demand t
  :init
  (let ((fonts
         '("all-the-icons"
           "file-icons"
           "FontAwesome"
           "github-octicons"
           "Material Icons")))
    (unless `(and
```

```
                (find-font (font-spec :name ,fonts)))
            (all-the-icons-install-fonts)))
        (setq spaceline-all-the-icons-icon-set-modified 'toggle)
        (setq spaceline-all-the-icons-icon-set-bookmark 'heart)
        (setq spaceline-all-the-icons-icon-set-flycheck-slim 'dots)
        (setq spaceline-all-the-icons-hide-long-buffer-path t)
        :config
        (spaceline-all-the-icons-theme)
        <<spaceline-modalka>>
        <<nyan>>
        )
```

8.4 Emojis!

8.4.1 emojify

Emojify renders a variety of strings as emojis, as well as providing some nice interactive functions to get emojis all over the place.

```
(use-package emojify
  :init
  (add-hook 'after-init-hook #'global-emojify-mode))
```

8.5 Fancy Pragmata Pro ligatures

```
;;; the-pragmata.el --- ligatures for Pragmata Pro

(setq prettify-symbols-unprettify-at-point 'right-edge)

(defconst pragmatapro-prettify-symbols-alist
  (mapcar (lambda (s)
            `(,(car s)
              .
              ,(vconcat
                (apply 'vconcat
                       (make-list
                        (- (length (car s)) 1)
                        (vector (decode-char 'ucs #X0020) '(Br . Bl))))
                (vector (decode-char 'ucs (cadr s))))))
          '(("[ERROR]"   #XE380)
            ("[DEBUG]"   #XE381)
```

```
("[INFO]"    #XE382)
("[WARN]"    #XE383)
("[WARNING]" #XE384)
("[ERR]"     #XE385)
("[FATAL]"   #XE386)
("[TRACE]"   #XE387)
("[FIXME]"   #XE388)
("[TODO]"    #XE389)
("[BUG]"     #XE38A)
("[NOTE]"    #XE38B)
("[HACK]"    #XE38C)
("[MARK]"    #XE38D)
("!!"        #XE900)
("!="        #XE901)
("!=="       #XE902)
("!!!"       #XE903)
("!"        #XE904)
("!"       #XE905)
("!>"        #XE906)
("!=<"       #XE907)
("#("        #XE920)
("#_"        #XE921)
("#{"        #XE922)
("#?"        #XE923)
("#>"        #XE924)
("##"        #XE925)
("#_("       #XE926)
("%="        #XE930)
("%>"        #XE931)
("%>%"       #XE932)
("%<%"       #XE933)
("&%"        #XE940)
("&&"        #XE941)
("&*"        #XE942)
("&+"        #XE943)
("&-"        #XE944)
("&/"        #XE945)
("&="        #XE946)
("&&&"       #XE947)
("&>"        #XE948)
```

```
("$>"        #XE955)
("***"       #XE960)
("*="        #XE961)
("*/"        #XE962)
("*>"        #XE963)
("++"        #XE970)
("+++"       #XE971)
("+="        #XE972)
("+>"        #XE973)
("++="       #XE974)
("--"        #XE980)
("-<"        #XE981)
("-<<"       #XE982)
("-="        #XE983)
("->"        #XE984)
("->>"       #XE985)
("---"       #XE986)
("-->"       #XE987)
("-+-"       #XE988)
("-\\/"      #XE989)
("-|>"       #XE98A)
("-<|"       #XE98B)
(".."        #XE990)
("..."       #XE991)
("..<"       #XE992)
(".>"        #XE993)
(".~"        #XE994)
(".="        #XE995)
("/*"        #XE9A0)
("//"        #XE9A1)
("/>"        #XE9A2)
("/="        #XE9A3)
("/=="       #XE9A4)
("///"       #XE9A5)
("/**"       #XE9A6)
(":::"       #XE9AF)
("::"        #XE9B0)
(":="        #XE9B1)
(":"        #XE9B2)
(":>"        #XE9B3)
```

```
(":=>"       #XE9B4)
(":("        #XE9B5)
(":-("       #XE9B6)
(":)"        #XE9B7)
(":-)"       #XE9B8)
(":/"        #XE9B9)
(":\\"       #XE9BA)
(":3"        #XE9BB)
(":D"        #XE9BC)
(":P"        #XE9BD)
(":>:"       #XE9BE)
(":<:"       #XE9BF)
("<$>"       #XE9C0)
("<*"        #XE9C1)
("<*>"       #XE9C2)
("<+>"       #XE9C3)
("<-"        #XE9C4)
("<<"        #XE9C5)
("<<<"       #XE9C6)
("<<="       #XE9C7)
("<="        #XE9C8)
("<=>"       #XE9C9)
("<>"        #XE9CA)
("<|>"       #XE9CB)
("<<-"       #XE9CC)
("<|"        #XE9CD)
("<=<"       #XE9CE)
("<~"        #XE9CF)
("<~~"       #XE9D0)
("<<~"       #XE9D1)
("<$"        #XE9D2)
("<+"        #XE9D3)
("<!>"       #XE9D4)
("<@>"       #XE9D5)
("<#>"       #XE9D6)
("<%>"       #XE9D7)
("<^>"       #XE9D8)
("<&>"       #XE9D9)
("<?>"       #XE9DA)
("<.>"       #XE9DB)
```

```
("</>"       #XE9DC)
("<\\>"      #XE9DD)
("<\">"      #XE9DE)
("<:>"       #XE9DF)
("<~>"       #XE9E0)
("<**>"      #XE9E1)
("<<^"       #XE9E2)
("<!"        #XE9E3)
("<@"        #XE9E4)
("<#"        #XE9E5)
("<%"        #XE9E6)
("<^"        #XE9E7)
("<&"        #XE9E8)
("<?"        #XE9E9)
("<."        #XE9EA)
("</"        #XE9EB)
("<\\"       #XE9EC)
("<\""       #XE9ED)
("<:"        #XE9EE)
("<->"       #XE9EF)
("<!--"      #XE9F0)
("<--"       #XE9F1)
("<~<"       #XE9F2)
("<==>"      #XE9F3)
("<|-"       #XE9F4)
("<<|"       #XE9F5)
("==<"       #XEA00)
("=="        #XEA01)
("==="       #XEA02)
("==>"       #XEA03)
("=>"        #XEA04)
("=~"        #XEA05)
("=>>"       #XEA06)
("=/="       #XEA07)
(""         #XEA10)
(""        #XEA11)
(":"        #XEA12)
(">-"        #XEA20)
(">="        #XEA21)
(">>"        #XEA22)
```

```
(">>-"       #XEA23)
(">=="       #XEA24)
(">>>"       #XEA25)
(">=>"       #XEA26)
(">>^"       #XEA27)
(">>|"       #XEA28)
(">!="       #XEA29)
("??"        #XEA40)
("?~"        #XEA41)
("?="        #XEA42)
("?>"        #XEA43)
("???"       #XEA44)
("?."        #XEA45)
("^="        #XEA48)
("^."        #XEA49)
("^?"        #XEA4A)
("^.."       #XEA4B)
("^<<"       #XEA4C)
("^>>"       #XEA4D)
("^>"        #XEA4E)
("\\\\"      #XEA50)
("\\>"       #XEA51)
("\\/-"      #XEA52)
("@>"        #XEA57)
("|="        #XEA60)
("||"        #XEA61)
("|>"        #XEA62)
("|||"       #XEA63)
("|+|"       #XEA64)
("|->"       #XEA65)
("|-->"      #XEA66)
("|=>"       #XEA67)
("|==>"      #XEA68)
("|>-"       #XEA69)
("|<<"       #XEA6A)
("||>"       #XEA6B)
("|>>"       #XEA6C)
("~="        #XEA70)
("~>"        #XEA71)
("~~>"       #XEA72)
```

```
            ("~>>"       #XEA73)
            ("[["        #XEA80)
            ("]]"        #XEA81)
            ("\">"       #XEA90)
            )))

(defun add-pragmatapro-prettify-symbols-alist ()
  (dolist (alias pragmatapro-prettify-symbols-alist)
    (push alias prettify-symbols-alist)))

(add-hook 'prog-mode-hook
          #'add-pragmatapro-prettify-symbols-alist)

(global-prettify-symbols-mode +1)

(provide 'the-pragmata)

;;; the-pragmata.el ends here
```

9 Navigation

9.1 Completion

9.1.1 Packages

1. Smex - Frecency for command history This package provides a simple mechanism for recording the user's command history so that it can be used to sort commands by usage. It is automatically used by Ivy. Note, however, that historian.el will hopefully replace smex soon, since it provides more functionality in a more elegant way. See 1, 2.

   ```
   (use-package smex)
   ```

2. flx - Fuzzy command matching This package provides a framework for sorting choices in a hopefully intelligent way based on what the user has typed in, using "fuzzy matching" (i.e. "ffap" matches "find-file-at-point"). See 1.

   ```
   (use-package flx)
   ```

3. Ivy - completing-read on steroids Ivy is a completion and narrowing framework. What does this mean? By default, Emacs has some basic tab-completion for commands, files, and so on. Ivy replaces this interface by showing a list of all the possible options, and narrowing it in an intelligent way (using smex and flx, if they are installed) as the user inputs a query. This is much faster.

   (a) Setup

      i. Lazy Loading We'll be making a few patches, so we want to make sure that the patches aren't loaded until `ivy` is.

      ```
      (el-patch-feature ivy)
      ```

      ii. Keymap We define a keymap for `ivy-mode` so we can remap buffer switching commands when `ivy` is active.

      ```
      (el-patch-defvar ivy-mode-map
            (let ((map (make-sparse-keymap)))
              (define-key map [remap switch-to-buffer]
                'ivy-switch-buffer)
              (define-key map [remap switch-to-buffer-other-window]
                'ivy-switch-buffer-other-window)
              map)
            "Keymap for ‘ivy-mode’.")
      ```

      iii. Minor Mode Patches We patch Ivy to be easily toggle-able, and to restore normal `completing-read` functionality if `ivy-mode` is disabled.

      ```
      (el-patch-define-minor-mode ivy-mode
        "Toggle Ivy mode on or off.
        Turn Ivy mode on if ARG is positive, off otherwise.
        Turning on Ivy mode sets ‘completing-read-function’ to
        ‘ivy-completing-read’.
        Global bindings:
        \\{ivy-mode-map}
        Minibuffer bindings:
        \\{ivy-minibuffer-map}"
        :group 'ivy
        :global t
        :keymap ivy-mode-map
        :lighter " ivy"
        (if ivy-mode
            (progn
      ```

```
        (setq completing-read-function 'ivy-completing-read)
        (el-patch-splice 2
          (when ivy-do-completion-in-region
            (setq completion-in-region-function 'ivy-completion-in-region)
      (setq completing-read-function 'completing-read-default)
      (setq completion-in-region-function 'completion--in-region)))
```

iv. Keybindings `ivy-resume` lets us jump back into the last completion session, which is pretty handy.

```
("C-x C-r" . ivy-resume)
```

A. Minibuffer bindings The behavior of Ivy in the minibuffer is a bit unintuitive, so we're gonna make it a bit more intuitive. In short, tab for navigation, return for interaction, and `C-j` to use the current candidate as is.

```
("TAB" . ivy-alt-done)
("<tab>" . ivy-alt-done)
("C-j" . ivy-immediate-done)
```

v. Fuzzy matching Fuzzy matching is nice almost everywhere, so we turn it on for all `ivy` completions except for `swiper` (text search), since fuzzy matching for text is weird. We also raise the `ivy-flx-limit` so that it will actually be used.

```
(setq ivy-re-builders-alist
      '((swiper . ivy--regex-plus)
        (t . ivy--regex-fuzzy)))
(setq ivy-flx-limit 2000)
```

(b) `use-package` declaration

```
(use-package ivy
  :demand t
  :init
  <<ivy-lazy-load>>
  <<ivy-keymap>>
  <<ivy-mode-patches>>
  (ivy-mode 1)
  :bind (
         <<ivy-global-bindings>>
         :map ivy-minibuffer-map
         <<ivy-minibuffer-bindings>>
         )
  :config
```

```
<<ivy-fuzzy>>
:delight ivy-mode)
```

4. Counsel - Ivy-ized standard Emacs commands Ivy is just a general-purpose completion framework. It can be used to generate improved versions of many stock Emacs commands. This is done by the Counsel library. (It also adds a few new commands, such as `counsel-git-grep`.)

   (a) Setup

      i. Bindings Counsel is a set of convenient commands based on Ivy meant to improve the built-in Emacs equivalents. We bind them to the normal Emacs keys so we can use Ivy nearly everywhere. We also have some other useful commands for finding and searching within Git repos, and a visual kill ring for yanking.

      ```
      ("M-x" . counsel-M-x)
      ("C-x C-f" . counsel-find-file)
      ("C-h f" . counsel-describe-function)
      ("C-h v" . counsel-describe-variable)
      ("C-h l" . counsel-load-library)
      ("C-h C-l" . counsel-find-library)
      ("C-h S" . counsel-info-lookup-symbol)
      ("C-x 8 RET" . counsel-unicode-char)
      ("C-c g" . counsel-git)
      ("C-c j" . counsel-git-grep)
      ("C-c k" . counsel-rg)
      ("M-y" . counsel-yank-pop)
      ```

      We also bind a key for use in expression buffers (like `eval-expression`) to give us history search.

      ```
      ("C-r" . counsel-expression-history)
      ```

      ii. Find file at point If there is a valid file at point, `counsel-find-file` will select that file by default.

      ```
      (setq counsel-find-file-at-point t)
      ```

   (b) `use-package` declaration

   ```
   (use-package counsel
     :bind (;; Use Counsel for common Emacs commands.
            <<counsel-bindings>>
            :map read-expression-map
            <<counsel-expression-bindings>>
   ```

```
        )
  :config
  <<counsel-ffap>>
)
```

5. Historian - Remember completion choices Remembers your choices in completion menus.

```
(use-package historian
  :demand t
  :config
  (historian-mode 1))
```

   (a) `ivy-historian` We use Historian to sort Ivy candidates by frecency+flx.

      i. Setup The only configuration we do here is to mess around with how Historian weights results.

```
(setq ivy-historian-freq-boost-factor 500)
(setq ivy-historian-recent-boost 500)
(setq ivy-historian-recent-decrement 50)
```

      ii. `use-package` declaration

```
(use-package ivy-historian
  :demand t
  :after ivy
  :config
  <<ivy-historian-weights>>
  (ivy-historian-mode 1))
```

6. Icicles - Sheesh Icicles is steroids for the steroids. I don't even know everything it does, so it's not on by default.

```
(use-package icicles
  :demand t)
```

9.2 Finding files

9.2.1 Dotfile shortcuts

1. Prefix key We define a custom prefix to provide shortcuts to edit common dotfiles, and another to edit the file in another window.

```
(defcustom the-find-dotfile-prefix
  (the-join-keys the-prefix "e")
  "Prefix key sequence for opening dotfiles.
  The function ‘the-register-dotfile’ creates a keybinding under
  this prefix, if you ask it to."
  :group 'the
  :type 'string)


(defcustom the-find-dotfile-other-window-prefix
  (the-join-keys the-prefix "o")
  "Prefix key sequencing for opening dotfiles in another window.
The function ‘the-register-dotfile’ creates a keybinding under
this prefix, if you ask it to.")
```

2. Registering dotfiles By providing a filename (relative to your `$HOME`), we define a function `the-find-<something>`, where <something> is a cleaned up version of the basename of the file. You can also specify a keybinding to be appended to the end of `the-find-*-prefix`, as well as a pretty filename if you'd rather not use the automatically generated one.

```
(defmacro the-register-dotfile
    (filename &optional keybinding pretty-filename)
  "Establish functions and keybindings to open a dotfile.
The FILENAME should be a path relative to the user's home
directory. Two interactive functions are created: one to find the
file in the current window, and one to find it in another window.
If KEYBINDING is non-nil, the first function is bound to that key
sequence after it is prefixed by ‘the-find-dotfile-prefix’,
and the second function is bound to the same key sequence, but
prefixed instead by
‘the-find-dotfile-other-window-prefix’ (provided that the two
prefixes are different).
This is best demonstrated by example. Suppose FILENAME is
\".emacs.d/init.el\", KEYBINDING is \"e i\",
‘the-find-dotfile-prefix’ is at its default value of \"M-T
e\", and ‘the-find-dotfile-other-window-prefix’ is at its
default value of \"M-T o\". Then ‘the-register-dotfile’ will
create the interactive functions ‘the-find-init-el’ and
```

```
‘the-find-init-el-other-window’, and it will bind them to the
key sequences \"M-T e e i\" and \"M-T o e i\" respectively.
If PRETTY-FILENAME, a string, is non-nil, then it will be used in
place of \"init-el\" in this example. Otherwise, that string will
be generated automatically from the basename of FILENAME."
  (let* ((bare-filename (replace-regexp-in-string ".*/" "" filename))
         (full-filename (concat "~/" filename))
         (defun-name (intern
                      (replace-regexp-in-string
                       "-+"
                       "-"
                       (concat
                        "the-find-"
                        (or pretty-filename
                            (replace-regexp-in-string
                             "[^a-z0-9]" "-"
                             bare-filename))))))
         (defun-other-window-name
           (intern
            (concat (symbol-name defun-name)
                    "-other-window")))
         (docstring (format "Edit file %s."
                            full-filename))
         (docstring-other-window
          (format "Edit file %s, in another window."
                  full-filename))
         (defun-form ‘(defun ,defun-name ()
                        ,docstring
                        (interactive)
                        (find-file ,full-filename)))
         (defun-other-window-form
           ‘(defun ,defun-other-window-name ()
              ,docstring-other-window
              (interactive)
              (find-file-other-window ,full-filename)))
         (full-keybinding
          (when keybinding
            (the-join-keys
             the-find-dotfile-prefix
             keybinding)))
```

```
     (full-other-window-keybinding
      (the-join-keys
       the-find-dotfile-other-window-prefix
       keybinding)))
  `(progn
     ,defun-form
     ,defun-other-window-form
     (bind-keys
      ,@(when full-keybinding
          `((,full-keybinding . ,defun-name)))
      ,@(when (and full-other-window-keybinding
                   (not (string=
                         full-keybinding
                         full-other-window-keybinding)))
          `((,full-other-window-keybinding
             . ,defun-other-window-name))))
     ;; Return the symbols for the two functions defined.
     (list ',defun-name ',defun-other-window-name))))
```

(a) Emacs

```
(the-register-dotfile ".emacs.d/init.el" "e i")
(the-register-dotfile ".emacs.d/init.local.el" "e l")
```

(b) Git

```
(the-register-dotfile ".gitconfig" "g c")
(the-register-dotfile ".gitexclude" "g e")
(the-register-dotfile ".gitconfig.local" "g l")
```

(c) Fish

```
(the-register-dotfile ".config/fish/config.fish" "f c")
```

(d) ChunkWM

```
(the-register-dotfile ".chunkwmrc" "w m")

(defun the-reload-chunkwm ()
  (interactive)
  (async-shell-command "sh ~/.chunkwmrc"))
```

(e) SKHD

```
(the-register-dotfile ".skhdrc" "h k")
```

9.2.2 Visiting files

1. Symlinks Follow symlinks when opening files. This has the concrete impact, for instance, that when you edit init.el with the shortcut provided by `the-register-dotfile` and then later do `find-file`, you will be in the THE repository instead of your home directory.

   ```
   (setq find-file-visit-truename t)
   ```

   Disable the warning "X and Y are the same file" which normally appears when you visit a symlinked file by the same name. (Doing this isn't dangerous, as it will just redirect you to the existing buffer.)

   ```
   (setq find-file-suppress-same-file-warnings t)
   ```

2. VC nonsense Disable Emacs' built-in version control handling. This improves performance and disables some annoying warning messages and prompts, especially regarding symlinks. I only use Magit, and the `vc` machinery does all kinds of annoying stuff with performance and warnings.

   ```
   (setq vc-handled-backends nil)
   ```

3. Directory hygiene Automatically create any nonexistent parent directories when finding a file. If the buffer for the new file is killed without being saved, then offer to delete the created directory or directories.

   ```
   (defun the--advice-find-file-automatically-create-directory
       (original-function filename &rest args)
     "Automatically create and delete parent directories of files.
   This is an ‘:override’ advice for ‘find-file’ and friends. It
   automatically creates the parent directory (or directories) of
   the file being visited, if necessary. It also sets a buffer-local
   variable so that the user will be prompted to delete the newly
   created directories if they kill the buffer without saving it."
     ;; The variable ‘dirs-to-delete’ is a list of the directories that
     ;; will be automatically created by ‘make-directory’. We will want
     ;; to offer to delete these directories if the user kills the buffer
     ;; without saving it.
   ```

```
(let ((dirs-to-delete ()))
  ;; If the file already exists, we don't need to worry about
  ;; creating any directories.
  (unless (file-exists-p filename)
    ;; It's easy to figure out how to invoke `make-directory',
    ;; because it will automatically create all parent directories.
    ;; We just need to ask for the directory immediately containing
    ;; the file to be created.
    (let* ((dir-to-create (file-name-directory filename))
           ;; However, to find the exact set of directories that
           ;; might need to be deleted afterward, we need to iterate
           ;; upward through the directory tree until we find a
           ;; directory that already exists, starting at the
           ;; directory containing the new file.
           (current-dir dir-to-create))
      ;; If the directory containing the new file already exists,
      ;; nothing needs to be created, and therefore nothing needs to
      ;; be destroyed, either.
      (while (not (file-exists-p current-dir))
        ;; Otherwise, we'll add that directory onto the list of
        ;; directories that are going to be created.
        (push current-dir dirs-to-delete)
        ;; Now we iterate upwards one directory. The
        ;; `directory-file-name' function removes the trailing slash
        ;; of the current directory, so that it is viewed as a file,
        ;; and then the `file-name-directory' function returns the
        ;; directory component in that path (which means the parent
        ;; directory).
        (setq current-dir (file-name-directory
                           (directory-file-name current-dir))))
      ;; Only bother trying to create a directory if one does not
      ;; already exist.
      (unless (file-exists-p dir-to-create)
        ;; Make the necessary directory and its parents.
        (make-directory dir-to-create 'parents))))
  ;; Call the original `find-file', now that the directory
  ;; containing the file to found exists. We make sure to preserve
  ;; the return value, so as not to mess up any commands relying on
  ;; it.
  (prog1 (apply original-function filename args)
```

```
      ;; If there are directories we want to offer to delete later, we
      ;; have more to do.
      (when dirs-to-delete
        ;; Since we already called ‘find-file’, we’re now in the buffer
        ;; for the new file. That means we can transfer the list of
        ;; directories to possibly delete later into a buffer-local
        ;; variable. But we pushed new entries onto the beginning of
        ;; ‘dirs-to-delete’, so now we have to reverse it (in order to
        ;; later offer to delete directories from innermost to
        ;; outermost).
        (setq-local the--dirs-to-delete (reverse dirs-to-delete))
        ;; Now we add a buffer-local hook to offer to delete those
        ;; directories when the buffer is killed, but only if it’s
        ;; appropriate to do so (for instance, only if the directories
        ;; still exist and the file still doesn’t exist).
        (add-hook ’kill-buffer-hook
                  #’the--kill-buffer-delete-directory-if-appropriate
                  ’append ’local)
        ;; The above hook removes itself when it is run, but that will
        ;; only happen when the buffer is killed (which might never
        ;; happen). Just for cleanliness, we automatically remove it
        ;; when the buffer is saved. This hook also removes itself when
        ;; run, in addition to removing the above hook.
        (add-hook ’after-save-hook
                  #’the--remove-kill-buffer-delete-directory-hook
                  ’append ’local)))))

;; Add the advice that we just defined.
(advice-add #’find-file :around
            #’the--advice-find-file-automatically-create-directory)

;; Also enable it for ‘find-alternate-file’ (C-x C-v).
(advice-add #’find-alternate-file :around
            #’the--advice-find-file-automatically-create-directory)

;; Also enable it for ‘write-file’ (C-x C-w).
(advice-add #’write-file :around
            #’the--advice-find-file-automatically-create-directory)

(defun the--kill-buffer-delete-directory-if-appropriate ()
```

```
  "Delete parent directories if appropriate.
This is a function for ‘kill-buffer-hook’. If
‘the--advice-find-file-automatically-create-directory’ created
the directory containing the file for the current buffer
automatically, then offer to delete it. Otherwise, do nothing.
Also clean up related hooks."
  (when (and
         ;; Stop if there aren’t any directories to delete (shouldn’t
         ;; happen).
         the--dirs-to-delete
         ;; Stop if ‘the--dirs-to-delete’ somehow got set to
         ;; something other than a list (shouldn’t happen).
         (listp the--dirs-to-delete)
         ;; Stop if the current buffer doesn’t represent a
         ;; file (shouldn’t happen).
         buffer-file-name
         ;; Stop if the buffer has been saved, so that the file
         ;; actually exists now. This might happen if the buffer were
         ;; saved without ‘after-save-hook’ running, or if the
         ;; ‘find-file’-like function called was ‘write-file’.
         (not (file-exists-p buffer-file-name)))
    (cl-dolist (dir-to-delete the--dirs-to-delete)
      ;; Ignore any directories that no longer exist or are malformed.
      ;; We don’t return immediately if there’s a nonexistent
      ;; directory, because it might still be useful to offer to
      ;; delete other (parent) directories that should be deleted. But
      ;; this is an edge case.
      (when (and (stringp dir-to-delete)
                 (file-exists-p dir-to-delete))
        ;; Only delete a directory if the user is OK with it.
        (if (y-or-n-p (format "Also delete directory ‘%s’? "
                              ;; The ‘directory-file-name’ function
                              ;; removes the trailing slash.
                              (directory-file-name dir-to-delete)))
            (delete-directory dir-to-delete)
          ;; If the user doesn’t want to delete a directory, then they
          ;; obviously don’t want to delete any of its parent
          ;; directories, either.
          (cl-return)))))
  ;; It shouldn’t be necessary to remove this hook, since the buffer
```

```
  ;; is getting killed anyway, but just in case...
  (the--remove-kill-buffer-delete-directory-hook))

(defun the--remove-kill-buffer-delete-directory-hook ()
  "Clean up directory-deletion hooks, if necessary.
This is a function for ‘after-save-hook’. Remove
‘the--kill-buffer-delete-directory-if-appropriate’ from
‘kill-buffer-hook’, and also remove this function from
‘after-save-hook’."
  (remove-hook 'kill-buffer-hook
               #'the--kill-buffer-delete-directory-if-appropriate
               'local)
  (remove-hook 'after-save-hook
               #'the--remove-kill-buffer-delete-directory-hook
               'local))
```

4. Save place... When you open a file, position the cursor at the same place as the last time you edited the file.

```
(save-place-mode 1)
```

   (a) ...and shut up about it Inhibit the message that is usually printed when the ‘saveplace’ file is written.

```
(el-patch-defun save-place-alist-to-file ()
  (let ((file (expand-file-name save-place-file))
        (coding-system-for-write 'utf-8))
    (with-current-buffer (get-buffer-create " *Saved Places*")
      (delete-region (point-min) (point-max))
      (when save-place-forget-unreadable-files
        (save-place-forget-unreadable-files))
      (insert (format ";;; -*- coding: %s -*-\n"
                      (symbol-name coding-system-for-write)))
      (let ((print-length nil)
            (print-level nil))
        (pp save-place-alist (current-buffer)))
      (let ((version-control
             (cond
              ((null save-place-version-control) nil)
              ((eq 'never save-place-version-control) 'never)
```

```
      ((eq 'nospecial save-place-version-control) version-control)
      (t
       t))))
(condition-case nil
    ;; Don't use write-file; we don't want this buffer to visit it.
    (write-region (point-min) (point-max) file
                  (el-patch-add nil 'nomsg))
  (file-error (message "Saving places: can't write %s" file)))
(kill-buffer (current-buffer))))))
```

9.2.3 Projects

1. Projectile Projectile keeps track of a "project" list, which is automatically added to as you visit files in Git repositories, Node.js projects, etc. It then provides commands for quickly navigating between and within these projects.

   (a) Setup

      i. Enable projectile globally Also, ignore the repos set up by `straight.el`.

      ```
      (projectile-mode +1)
      (defun the-projectile-ignore-projects (project-root)
        (f-descendant-of? project-root (f-join user-emacs-directory "straight/re
      (setq projectile-ignored-project-function #'the-projectile-ignore-projects
      ```

      ii. Directory-local indexing In case your `.projectile` file is pretty hairy, this allows us to alter the indexing method as a dirlocal.

      ```
      (defun the-projectile-indexing-method-p (method)
        "Non-nil if METHOD is a safe value for `projectile-indexing-method'."
        (memq method '(native alien)))

      (put 'projectile-indexing-method 'safe-local-variable
           #'the-projectile-indexing-method-p)
      ```

   (b) `use-package` declaration

   ```
   (use-package projectile
     :demand t
     :config
     <<global-projectile>>
     <<projectile-index>>
   )
   ```

2. Counsel Projectile Counsel is everywhere! This integrates Projectile commands and Ivy.

   (a) `use-package` declaration

```
(use-package counsel-projectile
  :init
  (setq projectile-switch-project-action #'counsel-projectile-find-file)
  :config
  (counsel-projectile-mode))
```

9.3 Search

9.3.1 Regular Expressions

1. Rx A prescription for your regex woes. Don't write obscure regex syntax. Describe the regex you want, then generate the bizarre incantation you need.

```
(use-package rx)
```

10 Writing

10.1 Org Mode Customization

10.1.1 Global Outline Mode

Outlines work for just about any structured text imaginable, from code to prose. If it's got something that Emacs thinks is a paragraph, it works. When you need a high-level overview, it's hard to beat this.

```
(define-globalized-minor-mode global-outline-minor-mode
  outline-minor-mode outline-minor-mode)

(global-outline-minor-mode +1)
```

10.1.2 Org

Org is a hugely expansive framework (a.k.a. collection of hacks) for organizing information, notes, tasks, calendars, and anything else related to Org-anization.

1. Setup

(a) Version Hack Because `straight.el` runs Org directly from a Git repo, the autoloads Org uses to identify its version are not generated in the way that it expects. This causes it to either a) fail to determine its version at all or b) incorrectly report the version of the built-in Org which ships with Emacs. This causes some issues down the line, so we have to trick Org. This is how we do it.

First, we have to get the Git version, here represented by a short hash of the current commit.

```
(defun the-org-git-version ()
  (let ((git-repo
         (f-join user-emacs-directory "straight/repos/org")))
    (s-trim (git-run "describe"
                     "--match=release\*"
                     "--abbrev=6"
                     "HEAD"))))

(defun the-org-release ()
  (let ((git-repo
         (f-join user-emacs-directory "straight/repos/org")))
    (s-trim (s-chop-prefix "release_"
                           (git-run "describe"
                                    "--match=release\*"
                                    "--abbrev=0"
                                    "HEAD")))))
```

Next, we need to define `org-git-version` and `org-release` eagerly.

```
<<org-version>>
<<org-release>>
(defalias #'org-git-version #'the-org-git-version)
(defalias #'org-release #'the-org-release)
(provide 'org-version)
```

(b) `org-tempo` In the most recent release of Org, the way easy template expansion (i.e., `<s[TAB]` expands to a `begin_src` block) was changed to use `tempo`, so we need to require this in order to keep this very convenient functionality in place.

```
(defun the-fix-easy-templates ()
  (require 'org-tempo))
```

```
(add-hook 'org-mode-hook 'the-fix-easy-templates)
```

(c) Todo Sequence We use an augmented set of todo states, including TODO, IN-PROGRESS, WAITING, and the done states DONE and CANCELED.

```
(setq org-todo-keywords
      '((sequence
         "BACKLOG(b!)"
         "TODO(t!)"
         "NEXT(n)"
         "IN-PROGRESS(i!)"
         "|"
         "DONE(d!)")
        (sequence
         "WAITING(w@/!)"
         "HOLD(h@/!)"
         "|"
         "CANCELED(c@)")
        (type
         "PHONE(p!)"
         "MEETING(m!)")))

(setq org-todo-keyword-faces
      (quote (("TODO" :foreground "red" :weight bold)
              ("NEXT" :foreground "blue" :weight bold)
              ("IN-PROGRESS" :foreground "red" :weight bold)
              ("DONE" :foreground "forest green" :weight bold)
              ("WAITING" :foreground "orange" :weight bold)
              ("HOLD" :foreground "magenta" :weight bold)
              ("CANCELED" :foreground "forest green" :weight bold)
              ("MEETING" :foreground "forest green" :weight bold)
              ("PHONE" :foreground "forest green" :weight bold))))

(setq org-todo-state-tags-triggers
      (quote (("CANCELED" ("CANCELED" . t))
              ("WAITING" ("WAITING" . t))
              ("HOLD" ("WAITING") ("HOLD" . t))
              (done ("WAITING") ("HOLD"))
              ("TODO" ("WAITING") ("CANCELLED") ("HOLD"))
```

```
                ("NEXT" ("WAITING") ("CANCELLED") ("HOLD"))
                ("DONE" ("WAITING") ("CANCELLED") ("HOLD")))))

(setq org-use-fast-todo-selection t)
(setq org-treat-S-cursor-todo-selection-as-state-change nil)
(setq org-archive-location (f-expand "archive/%s::* Archived Tasks" org-direc
```

(d) Bindings

First, we want to set up some recommended bindings as specified in the Org manual.

```
("C-c a" . org-agenda)
("C-c c" . org-capture)
("C-c l" . org-store-link)
("C-c b" . org-iswitchb)
```

First, we move the Org bindings for `org-shift*` from the `S-` prefix to `C-`.

```
("S-<left>" . nil)
("S-<right>" . nil)
("S-<up>" . nil)
("S-<down>" . nil)
("C-<left>" . org-shiftleft)
("C-<right>" . org-shiftright)
("C-<up>" . org-shiftup)
("C-<down>" . org-shiftdown)
```

By default, Org maps `org-(backward/forward)-paragraph`, but only maps it to the keys we overrode for shift up and down. We'll remap all instances so that our existing bindings for those functions will work as expected.

```
([remap backward-paragraph] . org-backward-paragraph)
([remap forward-paragraph] . org-forward-paragraph)
```

Finally, we'll set up a convenient binding for inserting headings.

```
("M-RET" . org-insert-heading)
```

(e) Settings `org-insert-headline` will split your content by default, which is pretty dumb. We therefore set it to create a new heading,

instead. We also activate `org-indent-mode` for more beautiful documents.

We also set Org exports to occur asynchronously whenever possible.

Finally, we set up refile to use outline path completion for easier refiling.

```
(setq org-insert-heading-respect-content t)
(add-hook 'org-mode-hook #'org-indent-mode)
(setq org-export-in-background t)
(setq org-refile-use-outline-path t
      org-outline-path-complete-in-steps nil)
(setq org-log-into-drawer t)
(setq org-special-ctrl-a/e t
      org-special-ctrl-k t)
(setq org-return-follows-link t)
```

(f) Tags We set up some useful tags we'd like available in any Org buffer.

```
(setq org-tag-persistent-alist
      '((:startgroup . nil)
        ("@work" . ?w)
        ("@home" . ?h)
        ("@phone" . ?p)
        ("@mail" . ?m)
        (:endgroup . nil)
        ("ansible" . ?a)
        ("epic" . ?e)
        ("linux" . ?l)
        ("noexport" . ?n)
        ("crypt" . ?c)
        ))
```

(g) Default Org Directory We stick our Org files in a new directory in the home directory by default.

```
(setq org-directory "~/org")
```

(h) Capture Templates

```
(setq org-capture-templates
      '(("t" "Todo" entry (file+headline "~/org/inbox.org" "Tasks")
```

```
 "* TODO %?\n %T\n  %i\n  %a")
("g" "Groceries" entry (file+headline "~/org/groceries.org" "Groceries
 "* %?\nEntered on %U\n  %i")
("w" "Work" entry (file+headline "~/org/work.org" "Tasks")
 "* TODO %?\n %T\n %i\n %a")
("h" "Home" entry (file+headline "~/org/home.org" "Tasks")
 "* TODO %?\n %i")))

(setq org-refile-targets
      '((org-agenda-files :maxlevel . 3)))
```

(i) Utilities

i. Recursively sort buffer entries alphabetically

```
(defun the-org-sort-ignore-errors ()
  (condition-case x
      (org-sort-entries nil ?a)
    (user-error)))

(defun the-org-sort-buffer ()
  "Sort all entries in the Org buffer recursively in alphabetical order."
  (interactive)
  (org-map-entries #'the-org-sort-ignore-errors))
```

ii. Archive dead tasks If tasks are marked DONE, and either have no deadline or the deadline has passed, archive it.

```
(defun the-org-past-entries ()
  (when (and (string= (org-get-todo-state) "DONE")
             (let ((deadline (org-entry-get (point) "DEADLINE")))
               (or (null deadline)
                   (time-less-p (org-time-string-to-time deadline)
                                (current-time)))))
    (org-archive-subtree)
    (setq org-map-continue-from (line-beginning-position))))


(defun the-org-archive-past ()
  "Archive DONE items with deadlines either missing or in the past."
  (interactive)
  (org-map-entries #'the-org-past-entries))
```

iii. Pretty bullets We use `org-bullets` to make our outlines prettier. There's some minor alignment weirdness with my font,

so I may need to specify the bullet codepoints, later.

```
(use-package org-bullets
  :init
  (add-hook 'org-mode-hook 'org-bullets-mode))
```

iv. Dropbox integration If `~/org/` doesn't exist, but `~/Dropbox/org` does, symlink the latter to the former.

```
(if (and
     (not (f-exists? org-directory))
     (f-directory? "~/Dropbox/org"))
    (f-symlink "~/Dropbox/org" org-directory))
```

2. `use-package` declaration

```
(use-package org
  :straight org-plus-contrib
  :demand t
  :bind (
         <<basic-bindings>>
         :map org-mode-map
         <<org-mode-bindings>>
         <<org-mode-remaps>>
         <<org-mode-heading>>
         )
  :init
  :config
  <<org-version-definitions>>
  <<org-dir>>
  <<org-capture>>
  <<org-requires>>
  <<org-bullets>>
  <<org-settings>>
  <<org-sort-buffer>>
  <<org-archive-past>>
  <<todo-states>>
  <<org-dropbox>>
  <<org-tags>>
  :delight
  (org-indent-mode)
  )
```

10.1.3 Org Agenda

Org Agenda is for generating a more useful consolidated summary of all or some of your tasks, according to their metadata.

1. Setup

   (a) Bindings Analogously to our bindings for regular org files, we'll also move things off of `S-` and onto `C-`.

   ```
   ("S-<up>" . nil)
   ("S-<down>" . nil)
   ("S-<left>" . nil)
   ("S-<right>" . nil)
   ("C-<left>" . org-agenda-do-date-earlier)
   ("C-<right>" . org-agenda-do-date-later)
   ```

   (b) Window Splitting We want Org Agenda to split the window into two tall windows, rather than two wide windows stacked.

   ```
   (defun the--advice-org-agenda-split-horizontally (org-agenda &rest args)
     "Make ‘org-agenda’ split horizontally, not vertically, by default.
     This is an ‘:around’ advice for ‘org-agenda’. It commutes with
     ‘the--advice-org-agenda-default-directory’."
     (let ((split-height-threshold nil))
       (apply org-agenda args)))

   (advice-add #’org-agenda :around
               #’the--advice-org-agenda-split-horizontally)
   ```

   (c) Default Directory If `org-directory` exists, set `default-directory` to its value in the agenda so that things like `find-file` work sensibly.

   ```
   (defun the--advice-org-agenda-default-directory
       (org-agenda &rest args)
     "If ‘org-directory’ exists, set ‘default-directory’ to it in the agenda.
     This is an ‘:around’ advice for ‘org-agenda’. It commutes with
     ‘the--advice-org-agenda-split-horizontally’."
     (let ((default-directory (if (f-exists? org-directory)
                                  org-directory
                                default-directory)))
   ```

```
        (apply org-agenda args)))

    (advice-add #'org-agenda :around
                #'the--advice-org-agenda-default-directory)
```

(d) Settings

```
    (setq org-agenda-files '("~/org"))
    (setq org-agenda-skip-scheduled-if-done t
          org-agenda-skip-deadline-if-done t)
```

2. `use-package` declaration

```
(use-package org-agenda
  :straight org-plus-contrib
  :demand t
  :bind (:map org-agenda-mode-map
         <<org-agenda-bindings>>
         )
  :init
  <<agenda-files>>
  :config
  <<agenda-window-split>>
  <<agenda-default-directory>>
  )
```

10.1.4 Org Encryption

```
(use-package org-crypt
  :straight org-plus-contrib
  :demand t
  :config
  (org-crypt-use-before-save-magic)
  (setq org-tags-exclude-from-inheritance '("crypt"))
  (setq org-crypt-key "17F07DF3086C4BBFA5799F38EF21DED4826AAFCF"))
```

10.1.5 Org Journal

Keeping a regular record of what's going on at work (programmer's journal) and at home (personal journal) can be a useful habit, so let's give it a shot.

```
(use-package org-journal
```

```
  :demand t
  :config
  (setq org-journal-dir (f-expand "journal" org-directory))
  (setq org-journal-enable-encryption t))
```

10.1.6 Context-Aware Capture and Agenda

```
(use-package org-context
  :demand t
  :config
  (setq org-context-capture-shortcut
        '((todo
           "t" "Todo"
           entry (file+headline place-holder "Todos")
           "* TODO %?\n OPENED: %U by %n\n FILE: %a")
          (question
           "q" "Question"
           entry (file+headline place-holder "Questions")
           "* QUESTION %?\n OPENED: %U by %n\n FILE: %a")))
  (org-context-activate))
```

10.1.7 Extra Export Packages

In order to correctly export Org files to certain formats, we need some additional tools.

1. `htmlize` Used to convert symbols and such to HTML equivalents.

   ```
   (use-package htmlize)
   ```

10.1.8 Org-mode Config Settings

Our config files live in `the-lib-directory`, but our org source files live in `the-org-lib-directory`. Unless I decide to start loading org files directly (which is doable if a touch annoying, at times), for now I want the `:tangle` attribute set for me automatically as long as I'm working on one of THE's lib files.

Additionally, I'd like to regenerate the documentation on save so things will always be up to date.

```
(defun the-in-the-org-lib-p ()
```

```
  (and (f-this-file)
       (f-child-of? (f-this-file) the-org-lib-directory)))

(defun the-update-doc ()
  "Update the readme."
  (interactive)
  (save-window-excursion
    (progn
      (find-file the-doc-source-file)
      (org-md-export-to-markdown)
      (org-latex-export-to-pdf))))


(defun the-org-lib-hook ()
  (if (the-in-the-org-lib-p)
      (progn
        (setq-local org-babel-default-header-args:emacs-lisp
                    `((:tangle . ,(f-expand (f-swap-ext (f-filename (f-this-file)) "el"
                      (:noweb . "yes"))))))

  (add-hook 'org-mode-hook 'the-org-lib-hook)
```

Finally, I'd like to automatically tangle the files on save.

```
(defun the-org-lib-tangle-hook ()
  (if (the-in-the-org-lib-p)
      (org-babel-tangle)))

(add-hook 'after-save-hook 'the-org-lib-tangle-hook)
```

10.1.9 org-tree-slide

```
(use-package org-tree-slide
  :config
  (org-tree-slide-presentation-profile)
  (defun the-presentation-start ()
    (text-scale-set 5)
    (setq org-confirm-babel-evaluate nil)
    (setq ns-use-native-fullscreen t)
    (disable-theme 'gruvbox)
    (load-theme 'leuven)
```

```
  (toggle-frame-fullscreen))
(defun the-presentation-stop ()
  (text-scale-set 0)
  (setq org-confirm-babel-evaluate t)
  (disable-theme 'leuven)
  (load-theme 'gruvbox)
  (setq ns-use-native-fullscreen nil))
(add-hook 'org-tree-slide-play-hook #'the-presentation-start)
(add-hook 'org-tree-slide-stop-hook #'the-presentation-stop)
)
```

10.2 Editing Prose

10.2.1 Flyspell

Flyspell is Flycheck but for spelling. Simple as.

```
(use-package flyspell
  :bind* (("M-T ] s" . flyspell-goto-next-error))
  :diminish (flyspell-mode . "φ"))
```

10.3 Formatting

10.3.1 Formatting Options

1. Formatting

   (a) Sanity Don't use tabs for indentation, even in deeply indented lines.

   ```
   (setq-default indent-tabs-mode nil)
   ```

   Sentences end with one space, not two. We're not French typographers, so cut it out.

   ```
   (setq sentence-end-double-space nil)
   ```

   80 columns is the correct line length. Fight me.

   ```
   (setq-default fill-column 80)
   ```

   (b) Whitespace Trim trailing whitespace on save. This will get rid of end-of-line whitespace, and reduce the number of blank lines at the end of the file to one.

We don't always want this (though I almost always do), so we create a variable which is set globally, but which can be overridden on a per-file or per-directory basis.

```
(defvar the-delete-trailing-whitespace t
  "If non-nil, delete trailing whitespace on save.")

(put 'the-delete-trailing-whitespace
     'safe-local-variable #'booleanp)
```

And now we have a little helper to delete whitespace according to our variable.

```
(defun the--maybe-delete-trailing-whitespace ()
  "Maybe delete trailing whitespace in buffer.
Trailing whitespace is only deleted if variable
‘the-delete-trailing-whitespace’ if non-nil."
  (when the-delete-trailing-whitespace
    (delete-trailing-whitespace)))
```

Now we make sure whitespace is (maybe) deleted on save.

```
(add-hook 'before-save-hook
          #'the--maybe-delete-trailing-whitespace)
```

Finally, always end files with a newline.

```
(setq require-final-newline t)
```

i. `long-lines-mode` We define a minor mode for configuring `whitespace-mode` to highlight long lines. Enabling the mode will highlight characters beyond the fill column (80 columns, by default).

```
(define-minor-mode the-long-lines-mode
  "When enabled, highlight long lines."
  nil nil nil
  (if the-long-lines-mode
      (progn
        (setq-local whitespace-style '(face lines-tail))
        (whitespace-mode 1))
    (whitespace-mode -1)
    (kill-local-variable 'whitespace-style)))
```

(c) Line Wrapping When editing text (i.e., not code), we want to automatically keep lines a reasonable length (<80 columns).

```
(add-hook 'text-mode-hook #'auto-fill-mode)
```

`fill-paragraph` is pretty good, but some structured markup (like markdown) doesn't always play nice. `filladapt` will fill in these gaps. However, we shut it off in Org because Org already has its own version of the functionality of `filladapt`, and they don't agree with each other.

```
(use-package filladapt
  :demand t
  :config
  (add-hook 'text-mode-hook #'filladapt-mode)
  (add-hook 'org-mode-hook #'turn-off-filladapt-mode))
```

Use an adaptive fill prefix when visually wrapping too-long lines. This means that if you have a line that is long enough to wrap around, the prefix (e.g. comment characters or indent) will be displayed again on the next visual line. We turn it on everywhere by lifting it up to a global minor mode.

```
(use-package adaptive-wrap
  :demand t
  :config
  (define-globalized-minor-mode global-adaptive-wrap-prefix-mode
    adaptive-wrap-prefix-mode adaptive-wrap-prefix-mode)

  (global-adaptive-wrap-prefix-mode))
```

(d) EditorConfig EditorConfig is a tool for establishing and maintaining consistent code style in editors and IDEs which support it (most of the major ones have a plugin).

```
(use-package editorconfig)
```

(e) Utilities Like `reverse-region`, but works characterwise rather than linewise.

```
(defun the-reverse-characters (beg end)
  "Reverse the characters in the region from BEG to END.
Interactively, reverse the characters in the current region."
```

```
(interactive "*r")
(insert
 (reverse
  (delete-and-extract-region
   beg end))))
```

10.3.2 Indentation

1. Aggressive Indent Assuming your indentation is consistent, this will keep it correct without any additional work.

   (a) Setup

      i. Local Variable Here, we set up `aggressive-indent-mode` as a variable we can set on a file- or directory-local level.

      ```
      (put 'aggressive-indent-mode 'safe-local-variable #'booleanp)
      ```

      ii. Slow mode We register `aggressive-indent` with our slow mode, allowing us to disabled reindentation on save for situations in which reindentation is expensive. Note that `aggressive-indent--proccess-cha` is not a typo. Or rather, it is, but it's in the actual package, not on us.

      ```
      (defun the-aggressive-indent-toggle-slow ()
        "Slow down ‘aggressive-indent’ by disabling reindentation on save.
      This is done in ‘the-slow-indent-mode’."
        (add-hook 'aggressive-indent-mode-hook
                  #'the-aggressive-indent-toggle-slow)
        (if (or the-slow-indent-mode (not aggressive-indent-mode))
            (remove-hook 'before-save-hook
                         #'aggressive-indent--proccess-changed-list-and-indent
                         'local)
          (add-hook 'before-save-hook
                    #'aggressive-indent--proccess-changed-list-and-indent
                    nil 'local)))

        (add-hook 'the-slow-indent-mode #'the-aggressive-indent-toggle-slow)
      ```

   (b) `use-package` declaration

   ```
   (use-package aggressive-indent
     :init
     <<agg-indent-local>>
     :config
   ```

```
<<agg-indent-slow>>
:delight (aggressive-indent-mode "AggrIndent"))
```

11 Reading

11.1 PDF Functionality

11.1.1 `pdf-tools`

DocView is the built-in PDF viewer in Emacs, but it's a bit meh. `pdf-tools` is significantly nicer, with much better support for in-document hyperlinks and fancy things like that. It does require compilation of an external library, though.

```
(use-package pdf-tools
  :init
  (pdf-tools-install)
  (setq pdf-view-midnight-colors '("#fe8019" . "#1d2021"))
  (add-hook 'pdf-view-mode-hook #'pdf-view-midnight-minor-mode))
```

12 Version Control

12.1 Git integration

12.1.1 Direct Interaction

For Elisp purposes, we occasionally need to get some piece of information from Git. We do this using `git.el`, a dead-simple git interaction library.

```
(use-package git
  :demand t)
```

12.1.2 Magit

Magit is one of the Emacs killer apps. It's a Git porcelain which makes interacting with Git intuitive, instructive, and quick.

```
(use-package magit
  :bind (;; Add important keybindings for Magit as described in the
         ;; manual [1].
         ;;
```

```
        ;; [1]: https://magit.vc/manual/magit.html#Getting-Started
        ("C-x g" . magit-status)
        ("C-x M-g" . magit-dispatch-popup))
  :init

  ;; Suppress the message we get about "Turning on
  ;; magit-auto-revert-mode" when loading Magit.
  (setq magit-no-message '("Turning on magit-auto-revert-mode..."))

  :config

  ;; Enable the C-c M-g shortcut to go to a popup of Magit commands
  ;; relevant to the current file.
  (global-magit-file-mode +1)

  ;; The default location for git-credential-cache is in
  ;; ~/.config/git/credential. However, if ~/.git-credential-cache/
  ;; exists, then it is used instead. Magit seems to be hardcoded to
  ;; use the latter, so here we override it to have more correct
  ;; behavior.
  (unless (file-exists-p "~/.git-credential-cache/")
    (let* ((xdg-config-home (or (getenv "XDG_CONFIG_HOME")
                                (expand-file-name "~/.config/")))
           (socket (expand-file-name "git/credential/socket" xdg-config-home)))
      (setq magit-credential-cache-daemon-socket socket))))
```

** git-commit

```
;; Allows editing Git commit messages from the command line (i.e. with
;; emacs or emacsclient as your core.editor).
(use-package git-commit
  :init

  ;; Lazy-load ‘git-commit’.

  (el-patch-feature git-commit)

  (el-patch-defconst git-commit-filename-regexp "/\\(\
     \\(\\(COMMIT\\|NOTES\\|PULLREQ\\|TAG\\)_EDIT\\|MERGE_\\|\\)MSG\
```

```
   \\|BRANCH_DESCRIPTION\\)\\'")

(el-patch-defun git-commit-setup-check-buffer ()
  (and buffer-file-name
       (string-match-p git-commit-filename-regexp buffer-file-name)
       (git-commit-setup)))

(el-patch-define-minor-mode global-git-commit-mode
  "Edit Git commit messages.
   This global mode arranges for ‘git-commit-setup’ to be called
   when a Git commit message file is opened.  That usually happens
   when Git uses the Emacsclient as $GIT_EDITOR to have the user
   provide such a commit message."
  :group 'git-commit
  :type 'boolean
  :global t
  :init-value t
  :initialize (lambda (symbol exp)
                (custom-initialize-default symbol exp)
                (when global-git-commit-mode
                  (add-hook 'find-file-hook 'git-commit-setup-check-buffer)))
  (if global-git-commit-mode
      (add-hook  'find-file-hook 'git-commit-setup-check-buffer)
    (remove-hook 'find-file-hook 'git-commit-setup-check-buffer)))

(global-git-commit-mode 1)

:config

;; Wrap summary at 50 characters as per [1].
;;
;; [1]: http://chris.beams.io/posts/git-commit/
(setq git-commit-summary-max-length 50))
```

1. Github Integration

```
   (use-package magithub
     :demand t
     :after magit
     :config (magithub-feature-autoinject t))
```

12.1.3 Get link to commit or source line

Occasionally it's useful to send a link to a specific commit or source line on <repo host of choice>, and this package makes that relatively easy.

```
(use-package git-link
  :demand t
  :bind (("C-c g l" . git-link)))
```

13 Programming Utilities

13.1 Syntax Checking

13.1.1 Flycheck

Flycheck provides a framework for in-buffer error and warning highlighting, or more generally syntax checking. It comes with a large number of checkers pre-defined, and other packages define more.

1. Settings
   (a) Enable Flycheck Globally Enable Flycheck in all buffers, but also allow for disabling it per-buffer.

   ```
   (global-flycheck-mode +1)
   (put 'flycheck-mode 'safe-local-variable #'booleanp)
   ```

   (b) Disable Flycheck in the modeline It's honestly more distracting than anything,

   ```
   (setq flycheck-mode-line nil)
   ```

2. `use-package` declaration

   ```
   (use-package flycheck
     :defer 3
     :config
     <<flycheck-global>>
     <<no-flycheck-modeline>>
     )
   ```

13.2 Auto-completion

13.2.1 Company Settings

- Show completions instantly, rather than after half a second.
- Show completions after typing three characters.
- Show a maximum of 10 suggestions. This is the default but I think it's best to be explicit.
- Always display the entire suggestion list onscreen, placing it above the cursor if necessary.
- Always display suggestions in the tooltip, even if there is only one. Also, don't display metadata in the echo area (this conflicts with ElDoc).
- Show quick-reference numbers in the tooltip (select a completion with M-1 through M-0).
- Prevent non-matching input (which will dismiss the completions menu), but only if the user interacts explicitly with Company.
- Company appears to override our settings in `company-active-map` based on `company-auto-complete-chars`. Turning it off ensures we have full control.
- Prevent Company completions from being lowercased in the completion menu. This has only been observed to happen for comments and strings in Clojure. (Although in general it will happen wherever the Dabbrev backend is invoked.)
- Only search the current buffer to get suggestions for `company-dabbrev` (a backend that creates suggestions from text found in your buffers). This prevents Company from causing lag once you have a lot of buffers open.
- Make company-dabbrev case-sensitive. Case insensitivity seems like a great idea, but it turns out to look really bad when you have domain-specific words that have particular casing.

```
(setq company-idle-delay 0)
(setq company-minimum-prefix-length 3)
```

```
(setq company-tooltip-limit 10)
(setq company-tooltip-minimum company-tooltip-limit)
(setq company-show-numbers t)
(setq company-auto-complete-chars nil)
(setq company-dabbrev-downcase nil)
(setq company-dabbrev-other-buffers nil)
(setq company-dabbrev-ignore-case nil)
```

1. Performance In case autocompletion is making Emacs drag, we add a toggle to slow it down.

```
(defun the-company-toggle-slow ()
  "Slow down ‘company’ by turning up the delays before completion starts.
This is done in ‘the-slow-autocomplete-mode’."
  (if the-slow-autocomplete-mode
      (progn
        (setq-local company-idle-delay 1)
        (setq-local company-minimum-prefix-length 3))
    (kill-local-variable 'company-idle-delay)
    (kill-local-variable 'company-minimum-prefix-length)))

(add-hook 'the-slow-autocomplete-mode-hook #'the-company-toggle-slow)
```

2. YaSnippet Hack Make it so that Company's keymap overrides Yasnippet's keymap when a snippet is active. This way, you can TAB to complete a suggestion for the current field in a snippet, and then TAB to move to the next field. Plus, C-g will dismiss the Company completions menu rather than cancelling the snippet and moving the cursor while leaving the completions menu on-screen in the same location.

```
(with-eval-after-load 'yasnippet
  ;; FIXME: this is all a horrible hack, can it be done with
  ;; ‘bind-key’ instead?
  ;;
  ;; This function translates the "event types" I get from
  ;; ‘map-keymap’ into things that I can pass to ‘lookup-key’
  ;; and ‘define-key’. It’s a hack, and I’d like to find a
  ;; built-in function that accomplishes the same thing while
  ;; taking care of any edge cases I might have missed in this
```

```
;; ad-hoc solution.
(defun the-normalize-event (event)
  "This function is a complete hack, do not use.
But in principle, it translates what we get from ‘map-keymap’
into what ‘lookup-key’ and ‘define-key’ want."
  (if (vectorp event)
      event
    (vector event)))

;; Here we define a hybrid keymap that delegates first to
;; ‘company-active-map’ and then to ‘yas-keymap’.
(setq the-yas-company-keymap
      ;; It starts out as a copy of ‘yas-keymap’, and then we
      ;; merge in all of the bindings from
      ;; ‘company-active-map’.
      (let ((keymap (copy-keymap yas-keymap)))
        (map-keymap
         (lambda (event company-cmd)
           (let* ((event (the-normalize-event event))
                  (yas-cmd (lookup-key yas-keymap event)))
             ;; Here we use an extended menu item with the
             ;; ‘:filter’ option, which allows us to
             ;; dynamically decide which command we want to
             ;; run when a key is pressed.
             (define-key keymap event
               ‘(menu-item
                 nil ,company-cmd :filter
                 (lambda (cmd)
                   ;; There doesn’t seem to be any obvious
                   ;; function from Company to tell whether or
                   ;; not a completion is in progress (à la
                   ;; ‘company-explicit-action-p’), so I just
                   ;; check whether or not ‘company-my-keymap’
                   ;; is defined, which seems to be good
                   ;; enough.
                   (if company-my-keymap
                       ’,company-cmd
                     ’,yas-cmd))))))
         company-active-map)
        keymap))
```

```
;; The function ‘yas--make-control-overlay’ uses the current
;; value of ‘yas-keymap’ to build the Yasnippet overlay, so to
;; override the Yasnippet keymap we only need to dynamically
;; rebind ‘yas-keymap’ for the duration of that function.
(defun the-advice-company-overrides-yasnippet
    (yas--make-control-overlay &rest args)
  "Allow ‘company’ to override ‘yasnippet’.
This is an ‘:around’ advice for ‘yas--make-control-overlay’."
  (let ((yas-keymap the-yas-company-keymap))
    (apply yas--make-control-overlay args)))

(advice-add #’yas--make-control-overlay :around
            #’the-advice-company-overrides-yasnippet))
```

13.2.2 Company

`company` provides an in-buffer autocompletion framework. It allows for packages to define backends that supply completion candidates, as well as optional documentation and source code. Then Company allows for multiple frontends to display the candidates, such as a tooltip menu. Company stands for "Complete Anything".

```
(defvar the-company-backends-global
  ’(company-capf
    company-files
    (company-dabbrev-code company-keywords)
    company-dabbrev)
  "Values for ‘company-backends’ used everywhere.
If ‘company-backends’ is overridden by The, then these
backends will still be included.")

(use-package company
  :demand t
  :config
  (company-tng-configure-default)
  <<company-config>>
  <<company-slow>>
  (global-company-mode +1)
  :delight company-mode)
```

13.2.3 Company Statistics

`company-statistics` adds usage-based sorting to Company completions. It is a goal to replace this package with `historian` or `prescient`.

```
(use-package company-statistics
  :demand t
  :config

  ;; Let's future-proof our patching here just in case we ever decide
  ;; to lazy-load company-statistics.
  (el-patch-feature company-statistics)

  ;; Disable the message that is normally printed when
  ;; `company-statistics' loads its statistics file from disk.
  (el-patch-defun company-statistics--load ()
    "Restore statistics."
    (load company-statistics-file 'noerror
          (el-patch-swap nil 'nomessage)
          'nosuffix))

  ;; Enable Company Statistics.
  (company-statistics-mode +1))
```

14 Languages

14.1 Common Lisp

14.1.1 Aggressive Indent

Enable aggressive indentation in all Lisp modes.

```
(add-hook 'lisp-mode-hook #'aggressive-indent-mode)
```

14.2 Emacs Lisp

14.2.1 Hooks

Enable ElDoc for Elisp buffers and the **scratch** buffer.

```
(add-hook 'emacs-lisp-mode-hook #'eldoc-mode)
```

Enable Aggressive Indent for Elisp buffers and the **scratch** buffer.

```
(add-hook 'emacs-lisp-mode-hook #'aggressive-indent-mode)
```

14.2.2 Fixes

1. Advised Function Noise Emacs barfs up a bunch of nonsense warnings every time a function is advised, and we do that a lot, so we'll just tell it to hush.

```
(setq ad-redefinition-action 'accept)
```

2. Keyword lists Keyword lists are indented kind of stupidly. To wit, by default they will be indented like this:

```
(:foo bar
      :baz quux)

(:foo bar
 :bar quux)

(el-patch-defun lisp-indent-function (indent-point state)
  "This function is the normal value of the variable `lisp-indent-function'.
The function `calculate-lisp-indent' calls this to determine
if the arguments of a Lisp function call should be indented specially.
INDENT-POINT is the position at which the line being indented begins.
Point is located at the point to indent under (for default indentation);
STATE is the `parse-partial-sexp' state for that position.
If the current line is in a call to a Lisp function that has a non-nil
property `lisp-indent-function' (or the deprecated `lisp-indent-hook'),
it specifies how to indent.  The property value can be:
* `defun', meaning indent `defun'-style
  (this is also the case if there is no property and the function
  has a name that begins with \"def\", and three or more arguments);
* an integer N, meaning indent the first N arguments specially
  (like ordinary function arguments), and then indent any further
  arguments like a body;
* a function to call that returns the indentation (or nil).
  `lisp-indent-function' calls this function with the same two arguments
  that it itself received.
This function returns either the indentation to use, or nil if the
Lisp function does not specify a special indentation."
  (el-patch-let (($cond (and (elt state 2)
                             (el-patch-wrap 1 1
```

```
                              (or (not (looking-at "\\sw\\|\\s_"))
                                  (looking-at ":")))))
                    ($then (progn
                             (if (not (> (save-excursion (forward-line 1) (point))
                                         calculate-lisp-indent-last-sexp))
                                 (progn (goto-char calculate-lisp-indent-last-sexp)
                                        (beginning-of-line)
                                        (parse-partial-sexp (point)
                                                            calculate-lisp-indent-las
                             ;; Indent under the list or under the first sexp on the
                             ;; line as calculate-lisp-indent-last-sexp.  Note that i
                             ;; thing on that line has to be complete sexp since we a
                             ;; inside the innermost containing sexp.
                             (backward-prefix-chars)
                             (current-column)))
                    ($else (let ((function (buffer-substring (point)
                                                             (progn (forward-sexp 1)
                                 method)
                             (setq method (or (function-get (intern-soft function)
                                                            'lisp-indent-function)
                                              (get (intern-soft function) 'lisp-inden
                             (cond ((or (eq method 'defun)
                                        (and (null method)
                                             (> (length function) 3)
                                             (string-match "\\`def" function)))
                                    (lisp-indent-defform state indent-point))
                                   ((integerp method)
                                    (lisp-indent-specform method state
                                                          indent-point normal-indent)
                                   (method
                                    (funcall method indent-point state))))))
(let ((normal-indent (current-column))
      (el-patch-add
        (orig-point (point))))
  (goto-char (1+ (elt state 1)))
  (parse-partial-sexp (point) calculate-lisp-indent-last-sexp 0 t)
  (el-patch-swap
    (if $cond
        ;; car of form doesn't seem to be a symbol
        $then
```

```
  $else)
(cond
 ;; car of form doesn't seem to be a symbol, or is a keyword
 ($cond $then)
 ((and (save-excursion
         (goto-char indent-point)
         (skip-syntax-forward " ")
         (not (looking-at ":")))
       (save-excursion
         (goto-char orig-point)
         (looking-at ":")))
  (save-excursion
    (goto-char (+ 2 (elt state 1)))
    (current-column)))
 (t $else))))))
```

14.2.3 Reloading the Init File

First, we define a customizable keybinding to reload our init file.

```
(defcustom the-reload-init-keybinding
  (the-join-keys the-prefix "r")
  "The keybinding for reloading init.el, as a string.
Nil means no keybinding is established."
  :group 'the
  :type 'string)
```

Now we define a function to actually do the reload and bind it to our key.

```
(defun the-reload-init ()
  "Reload init.el."
  (interactive)
  (straight-transaction
    (straight-mark-transaction-as-init)
    (message "Reloading init.el...")
    (load user-init-file nil 'nomessage)
    (message "Reloading init.el... done.")))

(bind-key the-reload-init-keybinding #'the-reload-init)
```

14.2.4 Evaluate an Elisp buffer

Other Lisp interaction modes (like CIDER and Geiser) provide a binding for evaluating a whole buffer. We add a similar binding for `eval-buffer`, as well as some sanity-checking so we don't evaluate the init file in a bad way.

```
(defun the-eval-buffer ()
  "Evaluate the current buffer as Elisp code."
  (interactive)
  (message "Evaluating %s..." (buffer-name))
  (straight-transaction
    (if (null buffer-file-name)
        (eval-buffer)
      (when (string= buffer-file-name user-init-file)
        (straight-mark-transaction-as-init))
      (load buffer-file-name nil 'nomessage)))
  (message "Evaluating %s... done." (buffer-name)))

(bind-key "C-c C-k" #'the-eval-buffer emacs-lisp-mode-map)
```

14.2.5 Rebind Find Commands

Add keybindings (`C-h C-f` and `C-h C-v`) for jumping to the source of Elisp functions and variables. Also, add a keybinding (`C-h C-o`) that performs the functionality of `M-.` only for Elisp, because the latter command is often rebound by other major modes. Note that this overrides the default bindings of `C-h C-f` (`view-emacs-FAQ`) and `C-h C-o` (`describe-distribution`), but I've never used those in 10 years of Emacsing.

```
(defun find-symbol (&optional symbol)
  "Same as `xref-find-definitions' but only for Elisp symbols."
  (interactive)
  (let ((xref-backend-functions '(elisp--xref-backend)))
    (if symbol
        (xref-find-definitions symbol)
      (call-interactively 'xref-find-definitions))))

(bind-keys
 ("C-h C-f" . find-function)
 ("C-h C-v" . find-variable)
 ("C-h C-o" . find-symbol))
```

14.2.6 Lisp Interaction Lighter

Show ‘lisp-interaction-mode’ as "ξι" instead of "Lisp Interaction" in the mode line.

```
(defun the--rename-lisp-interaction-mode ()
  (setq mode-name "ξι"))

(add-hook 'lisp-interaction-mode-hook
          #'the--rename-lisp-interaction-mode)
```

15 Performance

15.1 Performance Mode

Occasionally features like indentation and autocompletion are expensive, so we set up a minor mode to slow them down.

15.1.1 Modes

```
(define-minor-mode the-slow-indent-mode
  "Minor mode for when the indentation code is slow.
This prevents ‘aggressive-indent’ from indenting as frequently.")

(define-minor-mode the-slow-autocomplete-mode
  "Minor mode for when the autocompletion code is slow.
This prevents ‘company’ and ‘eldoc’ from displaying metadata as
quickly.")
```

16 Networking

16.1 Network Services

16.1.1 macOS TLS verification

TLS certs on macOS don't live anywhere that `gnutls` can see them, by default, so `brew install libressl` and we'll use those.

```
(the-with-operating-system macOS
  (with-eval-after-load 'gnutls
    (setq gnutls-verify-error t)
    (setq gnutls-min-prime-bits 3072)
    (add-to-list 'gnutls-trustfiles "/usr/local/etc/libressl/cert.pem")))
```

16.1.2 StackOverflow

Honestly, probably the most important package here. `M-x sx-authenticate`, provide a username and password, then get to overflowing.

```
(use-package sx)
```

16.1.3 Bug URL references

Allow setting the regexp for bug references from file-local or directory-local variables. CIDER does this in its files, for example.

```
(put 'bug-reference-bug-regexp 'safe-local-variable #'stringp)
```

16.1.4 Pastebin

`ix.io` is a slick little pastebin, and now we can use it in Emacs.

```
(use-package ix)
```

16.1.5 Browsing

`eww` is the wonderfully named Emacs Web Wowser, a text-based browser.

```
(use-package eww
  :bind* (("M-T g x" . eww)
          ("M-T g :" . eww-browse-with-external-browser)
          ("M-T g #" . eww-list-histories)
          ("M-T g {" . eww-back-url)
          ("M-T g }" . eww-forward-url))
  :config
  (progn
    (add-hook 'eww-mode-hook 'visual-line-mode)))
```

16.1.6 Steam

```
(use-package steam
  :init
  (setq steam-username "prooftechnique"))
```

17 Etc.

17.1 Miscellaneous Utilities

17.1.1 Eventually-obsolete Functions

These functions will become unnecessary in Emacs 26.1, which extends `map-put` to have a TESTFN argument.

```
(defun the-alist-set (key val alist &optional symbol)
  "Set property KEY to VAL in ALIST. Return new alist.
This creates the association if it is missing, and otherwise sets
the cdr of the first matching association in the list. It does
not create duplicate associations. By default, key comparison is
done with ‘equal’. However, if SYMBOL is non-nil, then ‘eq’ is
used instead.
This method may mutate the original alist, but you still need to
use the return value of this method instead of the original
alist, to ensure correct results."
  (if-let* ((pair (if symbol (assq key alist) (assoc key alist))))
      (setcdr pair val)
    (push (cons key val) alist))
  alist)

(defmacro the-alist-set* (key val alist &optional symbol)
  "Set property KEY to VAL in ALIST. Return new alist.
ALIST must be a literal symbol naming a variable holding an
alist. That variable will be re-set using ‘setq’. By default, key
comparison is done with ‘equal’. However, if SYMBOL is non-nil,
then ‘eq’ is used instead. See also ‘the-alist-set’."
  ‘(setq ,alist (the-alist-set ,key ,val ,alist ,symbol)))

(defun the-insert-after (insert-elt before-elt list &optional testfn)
  "Insert INSERT-ELT after BEFORE-ELT in LIST, returning copy of LIST.
The original LIST is not modified. If BEFORE-ELT is not in LIST,
it is inserted at the end. Element comparison is done with
TESTFN, which defaults to ‘eq’. See also ‘the-insert-before’
and ‘the-insert-after*’."
  (let ((testfn (or testfn #’eq)))
    (cond
     ((null list)
```

```
      (list insert-elt))
     ((funcall testfn before-elt (car list))
      (append (list (car list) insert-elt) (copy-sequence (cdr list))))
     (t (cons (car list)
              (the-insert-after
               insert-elt before-elt (cdr list) testfn))))))

(defmacro the-insert-after* (insert-elt before-elt list &optional testfn)
  "Insert INSERT-ELT after BEFORE-ELT in LIST, returning copy of LIST.
LIST must be a literal symbol naming a variable holding a list.
That variable will be re-set using `setq'. Element comparison is
done with TESTFN, which defaults to `eq'. See also
`the-insert-after' and `the-insert-before'."
  `(setq ,list (the-insert-after ,insert-elt ,before-elt ,list ,testfn)))

(defun the-insert-before (insert-elt after-elt lst &optional testfn)
  "Insert INSERT-ELT before AFTER-ELT in LIST, returning copy of LIST.
The original LIST is not modified. If BEFORE-ELT is not in LIST,
it is inserted at the beginning. Element comparison is done with
TESTFN, which defaults to `eq'. See also `the-insert-after'
and `the-insert-before*'."
  (nreverse (the-insert-after insert-elt after-elt (reverse lst) testfn)))

(defmacro the-insert-before* (insert-elt after-elt list &optional testfn)
  "Insert INSERT-ELT before AFTER-ELT in LIST, returning copy of LIST.
LIST must be a literal symbol naming a variable holding a list.
That variable will be re-set using `setq'. Element comparison is
done with TESTFN, which defaults to `eq'. See also
`the-insert-before' and `the-insert-after*'."
  `(setq ,list (the-insert-before ,insert-elt ,after-elt ,list ,testfn)))
```

17.1.2 Framework Identification

This gives us an easy way to check if the file we're editing is part of THE.

```
(defun the-managed-p (filename)
  "Return non-nil if FILENAME is managed by The.
This means that FILENAME is a symlink whose target is inside
`the-directory'."
  (and the-directory
```

```
(string-prefix-p the-directory (file-truename filename)
                 ;; The filesystem on macOS is case-insensitive
                 ;; but case-preserving, so we have to compare
                 ;; case-insensitively in that situation.
                 (eq the-operating-system 'macOS))))
```