# The Heretic's Emacs config

Jack Henahan

January 11, 2018

## Contents

# 1 Appearance

## 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.

## 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)
```

## 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)
```

## 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)
```

## 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))))
```

### 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.

### 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)
```

## 2 Auto-completion

### 2.1 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
  :init (company-tng-configure-default)
  :config
  <<company-config>>
  :delight company-mode)
```

## 2.2 Company Settings

Now that all the bindings are out of the way, we do the following:

- 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)

;; Register ‘company’ in ‘the-slow-autocomplete-mode’.

(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)

  ;; 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))

  ;; Turn on Company everywhere.
  (global-company-mode +1)
```

;; Package ‘company-statistics’ adds usage-based sorting to Company ;; completions. It is a goal to replace this package with ‘historian’ ;; or ‘prescient’. See [1] and [2]. ;; ;; [1]: `https://github.com/PythonNut/`

historian.el ;; [2]: https://github.com/raxod502/prescient.el

```
(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))
```

## 3 Binding Keys

### 3.1 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))) " "))
```

### 3.2 `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)
```

## 4 Syntax Checking

### 4.1 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.

#### 4.1.1 Settings

1. 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)
   ```

2. Disable Flycheck in the modeline It's honestly more distracting than anything,

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

#### 4.1.2 `use-package` declaration

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

## 5 Clipboard Integration

### 5.1 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)))
```

## 5.2 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)
```