Doom Emacs Configuration

The Methods, Management, and Menagerie of Madness

tecosaur

2021-02-06 07:56 UTC, 0c50e2a

Contents

Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do. — Donald Knuth

1 Intro

Customising an editor can be very rewarding . . . until you have to leave it. For years I have been looking for ways to avoid this pain. Then I discovered vim-anywhere, and found that it had an Emacs companion, emacs-anywhere. To me, this looked most attractive.

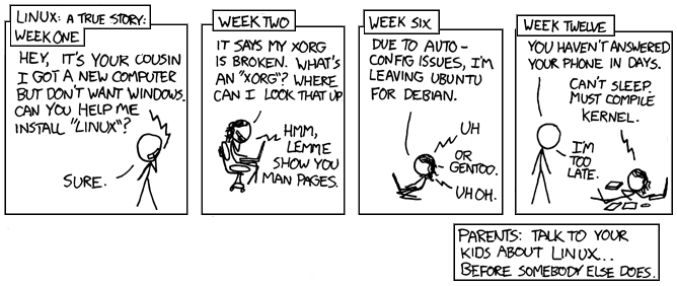
Separately, online I have seen the following statement enough times I think it's a catchphrase

> *Redditor 1: I just discovered this thing, isn't it cool.*
> *Redditor 2: Oh, there's an Emacs mode for that.*

This was enough for me to install Emacs, but I soon learned there are far more compelling reasons to keep using it.

I tried out the `spacemacs` distribution a bit, but it wasn't quite to my liking. Then I heard about `doom emacs` and thought I may as well give that a try. TLDR; it's great.

Now I've discovered the wonders of literate programming, and am becoming more settled by the day. This is both my config, and a cautionary tale (just replace "Linux" with "Emacs" in the comic below).



Cautionary This really is a true story, and she doesn't know I put it in my comic because her wifi hasn't worked for weeks.

1.1 Why Emacs?

Emacs is not a text editor, this is a common misnomer. It is far more apt to describe Emacs as *a Lisp machine providing a generic user-centric text manipulation environment*. That's quite a mouthful. In simpler terms one can think of Emacs as a platform for text-related applications. It's a vague and generic definition because Emacs itself is generic.

Good with text. How far does that go? A lot further than one initially thinks:

- Task planning
- File management
- Terminal emulation
- Email client
- Remote server tool
- Git frontend
- Web client/server
- and more...

Ideally, one may use Emacs as *the* interface to perform `input ¿ transform ¿ output` cycles, i.e. form a bridge between the human mind and information manipulation.

1.1.1 The enveloping editor

Emacs allows one to do more in one place than any other application. Why is this good?

- Enables one to complete tasks with a consistent, standard set of keybindings, GUI and editing methods — learn once, use everywhere
- Reduced context-switching
- Compressing the stages of a project — a more centralised workflow can progress with greater ease
- Integration between tasks previously relegated to different applications, but with a

common subject — e.g. linking to an email in a to-do list

Emacs can be thought of as a platform within which various elements of your workflow may settle, with the potential for rich integrations between them — a *life* IDE if you will.

Today, many aspects of daily computer usage are split between different applications which act like islands, but this often doesn't mirror how we *actually use* our computers. Emacs, if one goes down the rabbit hole, can give users the power to bridge this gap.

Figure 1: Some sample workflow integrations that can be used within Emacs

1.1.2 Some notably unique features

- Recursive editing
- Completely introspectable, with pervasive docstrings
- Mutable environment, which can be incrementally modified
- Functionality without applications
- Client-server separation allows for a daemon, giving near-instant perceived startup time.

1.1.3 Issues

- Emacs has irritating quirks
- Some aspects are showing their age (naming conventions, APIs)
- Emacs is (mostly) single-threaded, meaning that when something holds that thread up the whole application freezes
- A few other nuisances

1.1.4 Teach a man to fish...

> *Give a man a fish, and you feed him for a day. Teach a man to fish, and you feed him for a lifetime. — Anne Isabella*

Most popular editors have a simple and pretty settings interface, filled with check-boxes, selects, and the occasional text-box. This makes it easy for the user to pick between common desirable behaviours. To me this is now like *giving a man a fish*.

What if you want one of those 'check-box' settings to be only on in certain conditions? Some editors have workspace settings, but that requires you to manually set the value for *every single instance*. Urgh, what a pain.

What if you could set the value of that 'check-box' setting to be the result of an arbitrary expression evaluated for each file? This is where an editor like Emacs comes in. Configuration for Emacs isn't a list of settings in JSON etc. it's **an executable program which modifies the behaviour of the editor to suit your liking**. This is 'teaching a man to fish'.

Emacs is built in the same language you configure it in (Emacs Lisp, or elisp). It comes with a broad array of useful functions for text-editing, and Doom adds a few handy little convenience functions.

Want to add a keybinding to delete the previous line? It's as easy as

```
(map! "C-d"
      (cmd! (previous-line)
            (kill-line)
            (forward-line)))
```

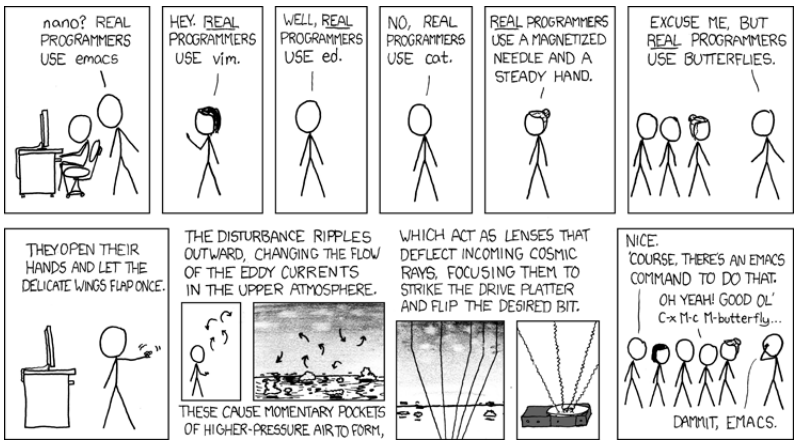
How about another example, say you want to be presented with a list of currently open *buffers* (think files, almost) when you split the window. It's as simple as

```
(defadvice! prompt-for-buffer (&rest _)
  :after 'window-split (switch-to-buffer))
```

Want to test it out? You don't need to save and restart, you can just *evaluate the expression* within your current Emacs instance and try it immediately! This editor is, after all, a Lisp interpreter.

Want to tweak the behaviour? Just re-evaluate your new version — it's a super-tight iteration loop.

1.2 Editor comparison



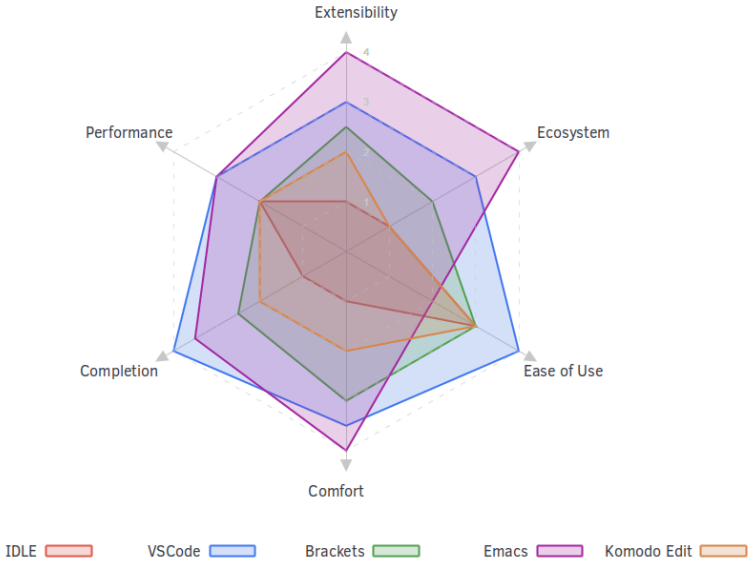
Real Programmers Real programmers set the universal constants at the start such that the universe evolves to contain the disk with the data they want.

Over the years I have tried out (spent at least a year using as my primary editor) the following applications

- Python IDLE
- Komodo Edit
- Brackets
- VSCode
- and now, Emacs

I have attempted to quantify aspects of my impressions of them below.

Editor	Extensibility	Ecosystem	Ease of Use	Comfort	Completion	Performance
IDLE	1	1	3	1	1	2
VSCode	3	3	4	3.5	4	3
Brackets	2.5	2	3	3	2.5	2
Emacs	4	4	2	4	3.5	3
Komodo Edit	2	1	3	2	2	2



1.3 Notes for the unwary adventurer

If you like the look of this, that's marvellous, and I'm really happy that I've made something which you may find interesting, however:

✤ **Warning**

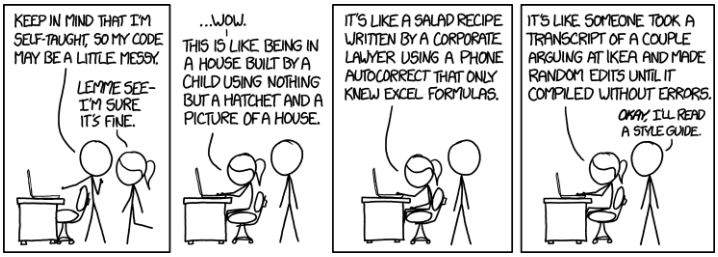
This config is *insidious*. Copying the whole thing blindly can easily lead to undesired effects. I recommend copying chunks instead.

If you are so bold as to wish to steal bits of my config (or if I upgrade and wonder why things aren't working), here's a list of sections which rely on external setup (i.e. outside of this config).

dictionary I've downloaded a custom SCOWL dictionary, which I use in ispell. If this causes issues, just delete the `(setq ispell-dictionary ...)` bit.

uni-units file I've got a file in `~/.org/.uni-units` which I use in `org-capture` If this causes issues, just remove the reference to that file in Capture and instances of `unit-prompt` used in `(doct ...)`

Oh, did I mention that I started this config when I didn't know any `elisp`, and this whole thing is a hack job? If you can suggest any improvements, please do so, no matter how much criticism you include I'll appreciate it :)



Code Quality I honestly didn't think you could even USE emoji in variable names. Or that there were so many different crying ones.

1.3.1 Extra Requirements

The lovely `doom doctor` is good at diagnosing most missing things, but here are a few extras.

- A LaTeX Compiler is required for the mathematics rendering performed in Org, and by CalcTeX.
- I use the Overpass font as a go-to sans serif. It's used as my `doom-variable-pitch-font` and in the graph generated by Roam. I have chosen it because it possesses a few characteristics I consider desirable, namely:
  - A clean, and legible style. Highway-style fonts tend to be designed to be clear at a glance, and work well with a thicker weight, and this is inspired by *Highway Gothic*.
  - It's slightly quirky. Look at the diagonal cut on stems for example. Helvetica is a masterful design, but I like a bit more pizzazz now and then.
- A few LSP servers. Take a look at init.el to see which modules have the `+lsp` flag.
- The Delta binary. It's packaged for some distributions but I installed it with

```
cargo install git-delta
```

- The `theme-magic` package requires the `wal` (`pywal`) executable. If this is packaged for you, great! If not, it's just a quick `pip install` away.

```
sudo python3 -m pip install pywal
```

1.4 Current Issues

1.4.1 Magit push in daemon

Quite often trying to push to a remote in the Emacs daemon produces as error like this:

```
128 git ¿ push -v origin refs/heads/master\:refs/heads/master
Pushing to git@github.com:tecosaur/emacs-config.git

fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

1.4.2 CalcTeX brings up compilation buffer

With my Calc hook, the first call of `M-x calc` brings up a compilation buffer from CalcTeX. I'm guessing this is from the compilation of the preamble / `.fmt` file.

1.4.3 Unread emails doesn't work across Emacs instances

It would be nice if it did, so that I could have the Emacs-daemon hold the active mu4e session, but still get that information. In this case I'd want to change the action to open the Emacs daemon, but it should be possible.

This would probably involve hooking into the daemon's modeline update function to write to a temporary file, and having a file watcher started in other Emacs instances, in a similar manner to Rebuild mail index while using mu4e.

2 Rudimentary configuration

Make this file run (slightly) faster with lexical binding (see this blog post for more info).

```
;;; config.el -*- lexical-binding: t; -*-
```

2.1 Personal Information

It's useful to have some basic personal information

```
(setq user-full-name "TEC"
      user-mail-address "tec@tecosaur.com")
```

Apparently this is used by `GPG`, and all sorts of other things.

Speaking of `GPG`, I want to use `~/.authsource.gpg` instead of the default in `~/.emacs.d`. Why? Because my home directory is already cluttered, so this won't make a difference, and I don't want to accidentaly purge this file (I have done `rm -rf~/.emac.d` before). I also want to cache as much as possible, as my home machine is pretty safe, and my laptop is shutdown a lot.

```
(setq auth-sources '("~/.authinfo.gpg")
      auth-source-cache-expiry nil) ; default is 7200 (2h)
```

2.2 Better defaults

2.2.1 Simple settings

Browsing the web and seeing angrybacon/dotemacs and comparing with the values shown by `SPC h v` and selecting what I thought looks good, I've ended up adding the following:

```
(setq-default
 delete-by-moving-to-trash t                      ; Delete files to trash
 window-combination-resize t                      ; take new window space from all
 ↪  other windows (not just current)
 x-stretch-cursor t)                              ; Stretch cursor to the glyph
 ↪  width

(setq undo-limit 80000000                         ; Raise undo-limit to 80Mb
      evil-want-fine-undo t                       ; By default while in insert all
      ↪  changes are one big blob. Be more granular
      auto-save-default t                         ; Nobody likes to loose work, I
      ↪  certainly don't
      truncate-string-ellipsis "¿")               ; Unicode ellispis are nicer than
      ↪  "...", and also save /precious/ space

(display-time-mode 1)                             ; Enable time in the mode-line
```

```
(if (equal "Battery status not available"
           (battery))
    (display-battery-mode 1)                    ; On laptops it's nice to know
    ↪  how much power you have
  (setq password-cache-expiry nil))             ; I can trust my desktops ...
  ↪  can't I? (no battery = desktop)

(global-subword-mode 1)                         ; Iterate through CamelCase words
```

2.2.2 Frame sizing

It's nice to control the size of new frames, when launching Emacs that can be done with `emacs -geometry 160x48`. After the font size adjustment during initialisation this works out to be `102x31`.

Thanks to hotkeys, it's easy for me to expand a frame to half/full-screen, so it makes sense to be conservative with the sizing of new frames.

Then, for creating new frames within the same Emacs instance, we'll just set the default to be something roughly 80% of that size.

```
(add-to-list 'default-frame-alist '(height . 24))
(add-to-list 'default-frame-alist '(width . 80))
```

2.2.3 Auto-customisations

By default changes made via a customisation interface are added to `init.el`. I prefer the idea of using a separate file for this. We just need to change a setting, and load it if it exists.

```
(setq-default custom-file (expand-file-name ".custom.el" doom-private-dir))
(when (file-exists-p custom-file)
  (load custom-file))
```

2.2.4 Windows

I find it rather handy to be asked which buffer I want to see after splitting the window. Let's make that happen. First, we'll enter the new window

```
(setq evil-vsplit-window-right t
      evil-split-window-below t)
```

Then, we'll pull up `ivy`

```
(defadvice! prompt-for-buffer (&rest _)
  :after '(evil-window-split evil-window-vsplit)
  (+ivy/switch-buffer))
```

Oh, and previews are nice

```
(setq +ivy-buffer-preview t)
```

Window rotation is nice, and can be found under `SPC w r` and `SPC w R`. *Layout* rotation is also nice though. Let's stash this under `SPC w SPC`, inspired by Tmux's use of `C-b SPC` to rotate windows.

We could also do with adding the missing arrow-key variants of the window navigation/swapping commands.

```
(map! :map evil-window-map
      "SPC" #'rotate-layout
      ;; Navigation
      "<left>"     #'evil-window-left
      "<down>"     #'evil-window-down
      "<up>"       #'evil-window-up
      "<right>"    #'evil-window-right
      ;; Swapping windows
      "C-<left>"       #'+evil/window-move-left
      "C-<down>"       #'+evil/window-move-down
      "C-<up>"         #'+evil/window-move-up
      "C-<right>"      #'+evil/window-move-right)
```

2.2.5 Buffer defaults

I'd much rather have my new buffers in `org-mode` than `fundamental-mode`, hence

```
;; (setq-default major-mode 'org-mode)
```

For some reason this + the mixed pitch hook causes issues with hydra and so I'll just need to resort to `SPC b o` for now.

2.3 Doom configuration

2.3.1 Modules

Doom has this lovely *modular configuration base* that takes a lot of work out of configuring Emacs. Each module (when enabled) can provide a list of packages to install (on `doom sync`) and configuration to be applied. The modules can also have flags applied to tweak their behaviour.

```
;;; init.el -*- lexical-binding: t; -*-

;; This file controls what Doom modules are enabled and what order they load in.
;; Press 'K' on a module to view its documentation, and 'gd' to browse its
↪  directory.

(doom! :completion
       <<doom-completion>>

       :ui
       <<doom-ui>>

       :editor
       <<doom-editor>>

       :emacs
       <<doom-emacs>>

       :term
       <<doom-term>>

       :checkers
       <<doom-checkers>>

       :tools
       <<doom-tools>>

       :os
       <<doom-os>>

       :lang
       <<doom-lang>>

       :email
       <<doom-email>>

       :app
       <<doom-app>>

       :config
       <<doom-config>>
```

```
      )
```

Structure As you may have noticed by this point, this is a literate configuration. Doom has good support for this which we access though the `literate` module.

While we're in the `:config` section, we'll use Dooms nicer defaults, along with the bindings and smartparens behaviour (the flags aren't documented, but they exist).

```
literate
(default +bindings +smartparens)
```

Interface There's a lot that can be done to enhance Emacs' capabilities. I reckon enabling half the modules Doom provides should do it.

```
(company                        ; the ultimate code completion backend
 +childframe)                   ; ... when your children are better than you
;;helm                          ; the *other* search engine for love and life
;;ido                           ; the other *other* search engine...
(ivy                            ; a search engine for love and life
 +icons                         ; ... icons are nice
 +prescient)                    ; ... I know what I want(ed)
```

```
;;deft                         ; notational velocity for Emacs
doom                            ; what makes DOOM look the way it does
doom-dashboard                  ; a nifty splash screen for Emacs
doom-quit                       ; DOOM quit-message prompts when you quit Emacs
;;fill-column                   ; a `fill-column' indicator
hl-todo                         ; highlight TODO/FIXME/NOTE/DEPRECATED/HACK/REVIEW
;;hydra                         ; quick documentation for related commands
;;indent-guides                 ; highlighted indent columns, notoriously slow
(ligatures +extra)              ; ligatures and symbols to make your code pretty again
;;minimap                       ; show a map of the code on the side
modeline                        ; snazzy, Atom-inspired modeline, plus API
nav-flash                       ; blink the current line after jumping
;;neotree                       ; a project drawer, like NERDTree for vim
ophints                         ; highlight the region an operation acts on
(popup                          ; tame sudden yet inevitable temporary windows
 +all                           ; catch all popups that start with an asterix
 +defaults)                     ; default popup rules
;;(tabs                         ; an tab bar for Emacs
;;  +centaur-tabs)              ; ... with prettier tabs
treemacs                        ; a project drawer, like neotree but cooler
;;unicode                       ; extended unicode support for various languages
vc-gutter                       ; vcs diff in the fringe
vi-tilde-fringe                 ; fringe tildes to mark beyond EOB
```

```
(window-select +numbers)      ; visually switch windows
workspaces                    ; tab emulation, persistence & separate workspaces
zen                           ; distraction-free coding or writing
```

```
(evil +everywhere)            ; come to the dark side, we have cookies
file-templates                ; auto-snippets for empty files
fold                          ; (nigh) universal code folding
(format +onsave)              ; automated prettiness
;;god                         ; run Emacs commands without modifier keys
;;lispy                       ; vim for lisp, for people who don't like vim
multiple-cursors              ; editing in many places at once
;;objed                       ; text object editing for the innocent
;;parinfer                    ; turn lisp into python, sort of
rotate-text                   ; cycle region at point between text candidates
snippets                      ; my elves. They type so I don't have to
;;word-wrap                   ; soft wrapping with language-aware indent
```

```
(dired +icons)                ; making dired pretty [functional]
electric                      ; smarter, keyword-based electric-indent
(ibuffer +icons)              ; interactive buffer management
(undo +tree)                  ; persistent, smarter undo for your inevitable mistakes
vc                            ; version-control and Emacs, sitting in a tree
```

```
;;eshell                      ; the elisp shell that works everywhere
;;shell                       ; simple shell REPL for Emacs
;;term                        ; basic terminal emulator for Emacs
vterm                         ; the best terminal emulation in Emacs
```

```
syntax                        ; tasing you for every semicolon you forget
spell                         ; tasing you for misspelling mispelling
grammar                       ; tasing grammar mistake every you make
```

```
ansible                       ; a crucible for infrastructure as code
debugger                      ; FIXME stepping through code, to help you add bugs
;;direnv                      ; be direct about your environment
;;docker                      ; port everything to containers
;;editorconfig                ; let someone else argue about tabs vs spaces
;;ein                         ; tame Jupyter notebooks with emacs
(eval +overlay)               ; run code, run (also, repls)
;;gist                        ; interacting with github gists
(lookup                       ; helps you navigate your code and documentation
 +dictionary                  ; dictionary/thesaurus is nice
 +docsets)                    ; ...or in Dash docsets locally
lsp                           ; Language Server Protocol
;;macos                       ; MacOS-specific commands
(magit                        ; a git porcelain for Emacs
 +forge)                      ; interface with git forges
make                          ; run make tasks from Emacs
```

```
;;pass                       ; password manager for nerds
pdf                         ; pdf enhancements
;;prodigy                    ; FIXME managing external services & code builders
rgb                         ; creating color strings
;;taskrunner                 ; taskrunner for all your projects
;;terraform                  ; infrastructure as code
;;tmux                       ; an API for interacting with tmux
upload                      ; map local to remote projects via ssh/ftp
```

```
tty                         ; improve the terminal Emacs experience
```

Language support We can be rather liberal with enabling support for languages as the associated packages/configuration are (usually) only loaded when first opening an associated file.

```
;;agda                      ; types of types of types of types...
;;cc                        ; C/C++/Obj-C madness
;;clojure                   ; java with a lisp
;;common-lisp               ; if you've seen one lisp, you've seen them all
;;coq                       ; proofs-as-programs
;;crystal                   ; ruby at the speed of c
;;csharp                    ; unity, .NET, and mono shenanigans
data                        ; config/data formats
;;(dart +flutter)           ; paint ui and not much else
;;elixir                    ; erlang done right
;;elm                       ; care for a cup of TEA?
emacs-lisp                  ; drown in parentheses
;;erlang                    ; an elegant language for a more civilized age
ess                         ; emacs speaks statistics
;;faust                     ; dsp, but you get to keep your soul
;;fsharp                    ; ML stands for Microsoft's Language
;;fstar                     ; (dependent) types and (monadic) effects and Z3
;;(go +lsp)                 ; the hipster dialect
(haskell +dante)            ; a language that's lazier than I am
;;hy                        ; readability of scheme w/ speed of python
;;idris                     ;
;;json                      ; At least it ain't XML
;;(java +meghanada)         ; the poster child for carpal tunnel syndrome
(javascript +lsp)           ; all(hope(abandon(ye(who(enter(here))))))
;;julia                     ; a better, faster MATLAB
;;kotlin                    ; a better, slicker Java(Script)
(latex                      ; writing papers in Emacs has never been so fun
 +latexmk                   ; what else would you use?
 +cdlatex                   ; quick maths symbols
 +fold)                     ; fold the clutter away nicities
;;lean                      ; proof that mathematicians need help
;;factor                    ; for when scripts are stacked against you
;;ledger                    ; an accounting system in Emacs
lua                         ; one-based indices? one-based indices
```

```
markdown                        ; writing docs for people to ignore
;;nim                          ; python + lisp at the speed of c
;;nix                          ; I hereby declare "nix geht mehr!"
;;ocaml                        ; an objective camel
(org                            ; organize your plain life in plain text
 +pretty                        ; yessss my pretties! (nice unicode symbols)
 +dragndrop                     ; drag & drop files/images into org buffers
 ;;+hugo                       ; use Emacs for hugo blogging
 +jupyter                       ; ipython/jupyter support for babel
 +pandoc                        ; export-with-pandoc support
 +gnuplot                       ; who doesn't like pretty pictures
 ;;+pomodoro                   ; be fruitful with the tomato technique
 +present                       ; using org-mode for presentations
 +roam)                         ; wander around notes
;;perl                          ; write code no one else can comprehend
;;php                           ; perl's insecure younger brother
;;plantuml                      ; diagrams for confusing people more
;;purescript                    ; javascript, but functional
(python +lsp +pyright)          ; beautiful is better than ugly
;;qt                            ; the 'cutest' gui framework ever
;;racket                        ; a DSL for DSLs
;;rest                          ; Emacs as a REST client
;;rst                           ; ReST in peace
;;(ruby +rails)                 ; 1.step {|i| p "Ruby is #{i.even? ? 'love' : 'life'}"}
(rust +lsp)                     ; Fe2O3.unwrap().unwrap().unwrap().unwrap()
;;scala                         ; java, but good
scheme                          ; a fully conniving family of lisps
sh                              ; she sells {ba,z,fi}sh shells on the C xor
;;sml                           ; no, the /other/ ML
;;solidity                      ; do you need a blockchain? No.
;;swift                         ; who asked for emoji variables?
;;terra                         ; Earth and Moon in alignment for performance.
web                             ; the tubes
yaml                            ; JSON, but readable
```

Everything in Emacs It's just too convenient being able to have everything in Emacs. I couldn't resist the Email and Feed modules.

```
(mu4e +org +gmail)
;;notmuch
;;(wanderlust +gmail)
```

```
;;calendar
irc                             ; how neckbeards socialize
(rss +org)                      ; emacs as an RSS reader
;;twitter                       ; twitter client https://twitter.com/vnought
```

2.3.2 Visual Settings

Font Face 'Fira Code' is nice, and 'Overpass' makes for a nice sans companion. We just need to fiddle with the font sizes a tad so that they visually match. Just for fun I'm trying out JetBrains Mono though. So far I have mixed feelings on it, some aspects are nice, but on others I prefer Fira.

```
(setq doom-font (font-spec :family "JetBrains Mono" :size 24)
      doom-big-font (font-spec :family "JetBrains Mono" :size 36)
      doom-variable-pitch-font (font-spec :family "Overpass" :size 24)
      doom-serif-font (font-spec :family "IBM Plex Mono" :weight 'light))
```

'Fira Code' is nice, and 'Overpass' makes for a nice sans companion. We just need to fiddle with the font sizes a tad so that they visually match. Just for fun I'm trying out JetBrains Mono though. So far I have mixed feelings on it, some aspects are nice, but on others I prefer Fira.
» emacs-lisp
(setq doom-font (font-spec :family "JetBrains Mono" :size 24)
      doom-big-font (font-spec :family "JetBrains Mono" :size 36)
      doom-variable-pitch-font (font-spec :family "Overpass" :size 24)
      doom-serif-font (font-spec :family "IBM Plex Mono" :weight 'light))
«

Theme and modeline `doom-one` is nice and all, but I find the `vibrant` variant nicer. Oh, and with the nice selection doom provides there's no reason for me to want the defaults.

```
(setq doom-theme 'doom-vibrant)
(delq! t custom-theme-load-path)
```

However, by default `red` text is used in the `modeline`, so let's make that orange so I don't feel like something's gone *wrong* when editing files.

```
(custom-set-faces!
  '(doom-modeline-buffer-modified :foreground "orange"))
```

While we're modifying the modeline, `LF UTF-8` is the default file encoding, and thus not worth noting in the modeline. So, let's conditionally hide it.

```
(defun doom-modeline-conditional-buffer-encoding ()
  "We expect the encoding to be LF UTF-8, so only show the modeline when this is not
  ↪ the case"
  (setq-local doom-modeline-buffer-encoding
```

```
              (unless (or (eq buffer-file-coding-system 'utf-8-unix)
                          (eq buffer-file-coding-system 'utf-8)))))

(add-hook 'after-change-major-mode-hook #'doom-modeline-conditional-buffer-encoding)
```

Miscellaneous Relative line numbers are fantastic for knowing how far away line numbers are, then `ESC 12 <UP>` gets you exactly where you think.

```
(setq display-line-numbers-type 'relative)
```

I'd like some slightly nicer default buffer names

```
(setq doom-fallback-buffer-name "¿ Doom"
      +doom-dashboard-name "¿ Doom")
```

There's a bug with the modeline in insert mode for org documents (issue), so

```
(custom-set-faces! '(doom-modeline-evil-insert-state :weight bold :foreground
↪  "#339CDB"))
```

2.3.3 Some helper macros

There are a few handy macros added by doom, namely

- `load!` for loading external `.el` files relative to this one
- `use-package!` for configuring packages
- `add-load-path!` for adding directories to the `load-path` where `Emacs` looks when you load packages with `require` or `use-package`
- `map!` for binding new keys

2.4 Other things

2.4.1 Editor interaction

Mouse buttons

```
(map! :n [mouse-8] #'better-jumper-jump-backward
      :n [mouse-9] #'better-jumper-jump-forward)
```

2.4.2 Window title

I'd like to have just the buffer name, then if applicable the project folder

```
(setq frame-title-format
      '(""
        (:eval
         (if (s-contains-p org-roam-directory (or buffer-file-name ""))
             (replace-regexp-in-string
              ".*/[0-9]*-?" "¿ "
              (subst-char-in-string ?_ ?  buffer-file-name))
           "%b"))
        (:eval
         (let ((project-name (projectile-project-name)))
           (unless (string= "-" project-name)
             (format (if (buffer-modified-p)  " ¿ %s" " ¿¿¿ %s") project-name))))))
```

For example when I open my config file it the window will be titled `config.org ¿ doom` then as soon as I make a change it will become `config.org ¿ doom`.

2.4.3 Splash screen

Emacs can render an image as the splash screen, and @MarioRicalde came up with a cracker! He's also provided me with a nice Emacs-style *E*, which is good for smaller windows. **@MarioRicalde** you have my sincere thanks, you're great!

By incrementally stripping away the outer layers of the logo one can obtain quite a nice resizing effect.

```
(defvar fancy-splash-image-template
  (expand-file-name "misc/splash-images/blackhole-lines-template.svg"
  ↪  doom-private-dir)
  "Default template svg used for the splash image, with substitutions from ")
(defvar fancy-splash-image-nil
  (expand-file-name "misc/splash-images/transparent-pixel.png" doom-private-dir)
  "An image to use at minimum size, usually a transparent pixel")
```

```
(setq fancy-splash-sizes
      `((:height 500 :min-height 50 :padding (0 . 4) :template ,(expand-file-name
      ↪  "misc/splash-images/blackhole-lines-0.svg" doom-private-dir))
        (:height 440 :min-height 42 :padding (1 . 4) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-0.svg" doom-private-dir))
        (:height 400 :min-height 38 :padding (1 . 4) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-1.svg" doom-private-dir))
        (:height 350 :min-height 36 :padding (1 . 3) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-2.svg" doom-private-dir))
        (:height 300 :min-height 34 :padding (1 . 3) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-3.svg" doom-private-dir))
        (:height 250 :min-height 32 :padding (1 . 2) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-4.svg" doom-private-dir))
        (:height 200 :min-height 30 :padding (1 . 2) :template ,(expand-file-name
        ↪  "misc/splash-images/blackhole-lines-5.svg" doom-private-dir))
        (:height 100 :min-height 24 :padding (1 . 2) :template ,(expand-file-name
        ↪  "misc/splash-images/emacs-e-template.svg" doom-private-dir))
        (:height 0   :min-height 0  :padding (0 . 0) :file
        ↪  ,fancy-splash-image-nil)))

(defvar fancy-splash-sizes
  `((:height 500 :min-height 50 :padding (0 . 2))
    (:height 440 :min-height 42 :padding (1 . 4))
    (:height 330 :min-height 35 :padding (1 . 3))
    (:height 200 :min-height 30 :padding (1 . 2))
    (:height 0   :min-height 0  :padding (0 . 0) :file ,fancy-splash-image-nil))
  "list of plists with the following properties
        :height the height of the image
        :min-height minimum `frame-height' for image
        :padding `+doom-dashboard-banner-padding' to apply
        :template non-default template file
        :file file to use instead of template")

(defvar fancy-splash-template-colours
  '(("$colour1" . keywords) ("$colour2" . type) ("$colour3" . base5) ("$colour4" .
  ↪  base8))
  "list of colour-replacement alists of the form (\"$placeholder\" . 'theme-colour)
  ↪  which applied the template")

(unless (file-exists-p (expand-file-name "theme-splashes" doom-cache-dir))
  (make-directory (expand-file-name "theme-splashes" doom-cache-dir) t))

(defun fancy-splash-filename (theme-name height)
  (expand-file-name (concat (file-name-as-directory "theme-splashes")
                            theme-name
                            "-" (number-to-string height) ".svg")
                    doom-cache-dir))

(defun fancy-splash-clear-cache ()
  "Delete all cached fancy splash images"
  (interactive)
  (delete-directory (expand-file-name "theme-splashes" doom-cache-dir) t)
```

```
  (message "Cache cleared!"))

(defun fancy-splash-generate-image (template height)
  "Read TEMPLATE and create an image if HEIGHT with colour substitutions as
       described by `fancy-splash-template-colours' for the current theme"
  (with-temp-buffer
    (insert-file-contents template)
    (re-search-forward "$height" nil t)
    (replace-match (number-to-string height) nil nil)
    (dolist (substitution fancy-splash-template-colours)
      (goto-char (point-min))
      (while (re-search-forward (car substitution) nil t)
        (replace-match (doom-color (cdr substitution)) nil nil)))
    (write-region nil nil
                  (fancy-splash-filename (symbol-name doom-theme) height) nil nil)))

(defun fancy-splash-generate-images ()
  "Perform `fancy-splash-generate-image' in bulk"
  (dolist (size fancy-splash-sizes)
    (unless (plist-get size :file)
      (fancy-splash-generate-image (or (plist-get size :file)
                                       (plist-get size :template)
                                       fancy-splash-image-template)
                                   (plist-get size :height)))))

(defun ensure-theme-splash-images-exist (&optional height)
  (unless (file-exists-p (fancy-splash-filename
                          (symbol-name doom-theme)
                          (or height
                              (plist-get (car fancy-splash-sizes) :height))))
    (fancy-splash-generate-images)))

(defun get-appropriate-splash ()
  (let ((height (frame-height)))
    (cl-some (lambda (size) (when (>= height (plist-get size :min-height)) size))
             fancy-splash-sizes)))

(setq fancy-splash-last-size nil)
(setq fancy-splash-last-theme nil)
(defun set-appropriate-splash (&rest _)
  (let ((appropriate-image (get-appropriate-splash)))
    (unless (and (equal appropriate-image fancy-splash-last-size)
                 (equal doom-theme fancy-splash-last-theme))
    (unless (plist-get appropriate-image :file)
      (ensure-theme-splash-images-exist (plist-get appropriate-image :height)))
    (setq fancy-splash-image
          (or (plist-get appropriate-image :file)
              (fancy-splash-filename (symbol-name doom-theme) (plist-get
              ↪  appropriate-image :height))))
    (setq +doom-dashboard-banner-padding (plist-get appropriate-image :padding))
    (setq fancy-splash-last-size appropriate-image)
    (setq fancy-splash-last-theme doom-theme)
    (+doom-dashboard-reload)))
```

```
(add-hook 'window-size-change-functions #'set-appropriate-splash)
(add-hook 'doom-load-theme-hook #'set-appropriate-splash)
```

2.4.4 Systemd daemon

For running a systemd service for a Emacs server I have the following

```
[Unit]
Description=Emacs server daemon
Documentation=info:emacs man:emacs(1) https://gnu.org/software/emacs/

[Service]
Type=forking
ExecStart=sh -c 'emacs --daemon && emacsclient -c --eval "(delete-frame)"'
ExecStop=/usr/bin/emacsclient --no-wait --eval "(progn (setq kill-emacs-hook nil)
↪  (kill emacs))"
Restart=on-failure

[Install]
WantedBy=default.target
```

which is then enabled by

```
systemctl --user enable emacs.service
```

For some reason if a frame isn't opened early in the initialisation process, the daemon doesn't seem to like opening frames later — hence the `&& emacsclient` part of the `ExecStart` value.

It can now be nice to use this as a 'default app' for opening files. If we add an appropriate desktop entry, and enable it in the desktop environment.

```
[Desktop Entry]
Name=Emacs client
GenericName=Text Editor
Comment=A flexible platform for end-user applications
MimeType=text/english;text/plain;text/x-makefile;text/x-c++hdr;text/x-c++src;text/x-
↪  chdr;text/x-csrc;text/x-java;text/x-moc;text/x-pascal;text/x-tcl;text/x-
↪  tex;application/x-shellscript;text/x-c;text/x-c++;
Exec=emacsclient -create-frame --alternate-editor="" --no-wait %F
Icon=emacs
Type=Application
Terminal=false
Categories=TextEditor;Utility;
StartupWMClass=Emacs
Keywords=Text;Editor;
X-KDE-StartupNotify=false
```

When the daemon is running, I almost always want to do a few particular things with it, so I may as well eat the load time at startup. We also want to keep `mu4e` running.

It would be good to start the IRC client (`circe`) too, but that seems to have issues when started in a non-graphical session.

```
(defun greedily-do-daemon-setup ()
  (require 'org)
  (when (require 'mu4e nil t)
    (setq mu4e-confirm-quit t)
    (setq +mu4e-lock-greedy t)
    (setq +mu4e-lock-relaxed t)
    (+mu4e-lock-add-watcher)
    (when (+mu4e-lock-available t)
      (mu4e~start)))
  (when (require 'elfeed nil t)
    (run-at-time nil (* 8 60 60) #'elfeed-update)))

(when (daemonp)
  (add-hook 'emacs-startup-hook #'greedily-do-daemon-setup))
```

2.4.5 Emacs client wrapper

I frequently want to make use of Emacs while in a terminal emulator. To make this easier, I can construct a few handy aliases.

However, a little convenience script in `~/.local/bin` can have the same effect, be available beyond the specific shell I plop the alias in, then also allow me to add a few bells and whistles — namely:

- Accepting stdin by putting it in a temporary file and immediately opening it.
- Guessing that the `tty` is a good idea when `$DISPLAY` is unset (relevant with SSH sessions, among other things).
- With a whiff of 24-bit color support, sets `TERM` variable to a `terminfo` that (probably) announces 24-bit color support.
- Changes GUI `emacsclient` instances to be non-blocking by default (`--no-wait`), and instead take a flag to suppress this behaviour (`-w`).

I would use `sh`, but using arrays for argument manipulation is just too convenient, so I'll raise the requirement to `bash`. Since arrays are the only 'extra' compared to `sh`, other shells like `csh` etc. should work too.

```
#!/usr/bin/env bash
force_tty=false
force_wait=false
stdin_mode=""

args=()

while :; do
    case "$1" in
        -t | -nw | --tty)
            force_tty=true
            shift ;;
        -w | --wait)
            force_wait=true
            shift ;;
        -m | --mode)
            stdin_mode=" ($2-mode)"
            shift 2 ;;
        -h | --help)
```

```
            echo -e "\033[1mUsage: e [-t] [-m MODE] [OPTIONS] FILE [-]\033[0m
                Emacs client convenience wrapper.
                \033[1mOptions:\033[0m
                \033[0;34m-h, --help\033[0m            Show this message
                \033[0;34m-t, -nw, --tty\033[0m        Force terminal mode
                \033[0;34m-w, --wait\033[0m            Don't supply
                \033[0;34m--no-wait\033[0m to graphical emacsclient
                \033[0;34m-\033[0m                     Take \033[0;33mstdin\033[0m
                (when last argument)
                \033[0;34m-m MODE, --mode MODE\033[0m  Mode to open
                \033[0;33mstdin\033[0m with
                Run \033[0;32memacsclient --help\033[0m to see help for the
            ↪  emacsclient."
            exit 0 ;;
        --*=*)
            set -- "$@" "${1%%=*}" "${1#*=}"
            shift ;;
        *)
            if [ "$#" = 0 ]; then
                break; fi
            args+=("$1")
            shift ;;
    esac
done

if [ ! "${#args[*]}" = 0 ] && [ "${args[-1]}" = "-" ]; then
    unset 'args[-1]'
    TMP="$(mktemp /tmp/emacsstdin-XXX)"
    cat > "$TMP"
    args+=(--eval "(let ((b (generate-new-buffer \"*stdin*\"))) (switch-to-buffer b)
    ↪  (insert-file-contents \"$TMP\") (delete-file \"$TMP\")${stdin_mode})")
fi

if [ -z "$DISPLAY" ] || $force_tty; then
    # detect terminals with sneaky 24-bit support
    if { [ "$COLORTERM" = truecolor ] || [ "$COLORTERM" = 24bit ]; } \
        && [ "$(tput colors 2>/dev/null)" -lt 257 ]; then
        if echo "$TERM" | grep -q "^\w\+-[0-9]"; then
            termstub="${TERM%%-*}"; else
            termstub="${TERM#*-}"; fi
        if infocmp "$termstub-direct" >/dev/null 2>&1; then
            TERM="$termstub-direct"; else
            TERM="xterm-direct"; fi # should be fairly safe
    fi
    emacsclient --tty -create-frame --alternate-editor="" "${args[@]}"
else
    if ! $force_wait; then
        args+=(--no-wait); fi
    emacsclient -create-frame --alternate-editor="" "${args[@]}"
fi
```

Now, to set an alias to use e with magit, and then for maximum laziness we can set

aliases for the terminal-forced variants.

```
alias m='e --eval "(progn (magit-status) (delete-other-windows))"'
alias mt="m -t"
alias et="e -t"
```

3 Package loading

This file shouldn't be byte compiled.

```
;; -*- no-byte-compile: t; -*-
```

3.1 Loading instructions

This is where you install packages, by declaring them with the `package!` macro, then running `doom refresh` on the command line. You'll need to restart Emacs for your changes to take effect! Or at least, run `M-x doom/reload`.

WARNING: Don't disable core packages listed in `~/.emacs.d/core/packages.el`. Doom requires these, and disabling them may have terrible side effects.

3.1.1 Packages in MELPA/ELPA/emacsmirror

To install `some-package` from MELPA, ELPA or emacsmirror:

```
(package! some-package)
```

3.1.2 Packages from git repositories

To install a package directly from a particular repo, you'll need to specify a `:recipe`. You'll find documentation on what `:recipe` accepts here:

```
(package! another-package
  :recipe (:host github :repo "username/repo"))
```

If the package you are trying to install does not contain a `PACKAGENAME.el` file, or is located in a subdirectory of the repo, you'll need to specify `:files` in the `:recipe`:

```
(package! this-package
  :recipe (:host github :repo "username/repo"
           :files ("some-file.el" "src/lisp/*.el")))
```

3.1.3 Disabling built-in packages

If you'd like to disable a package included with Doom, for whatever reason, you can do so here with the `:disable` property:

```
(package! builtin-package :disable t)
```

You can override the recipe of a built in package without having to specify all the properties for `:recipe`. These will inherit the rest of its recipe from Doom or MELPA/ELPA/Emacsmirror:

```
(package! builtin-package :recipe (:nonrecursive t))
(package! builtin-package-2 :recipe (:repo "myfork/package"))
```

Specify a `:branch` to install a package from a particular branch or tag. This is required for some packages whose default branch isn't 'master' (which our package manager can't deal with; see raxod502/straight.el#279)

```
(package! builtin-package :recipe (:branch "develop"))
```

3.2 General packages

3.2.1 Window management

```
(package! rotate :pin "4e9ac3ff800880bd9b705794ef0f7c99d72900a6")
```

3.2.2 Fun

Sometimes one just wants a little fun. XKCD comics are fun.

```
(package! xkcd :pin "66e928706fd660cfdab204c98a347b49c4267bdf")
```

Every so often, you want everyone else to *know* that you're typing, or just to amuse oneself. Introducing: typewriter sounds!

```
(package! selectric-mode :pin "1840de71f7414b7cd6ce425747c8e26a413233aa")
```

Hey, let's get the weather in here while we're at it. Unfortunately this seems slightly unmaintained (few open bugfix PRs) so let's roll our own version.

```
(package! wttrin :recipe (:local-repo "lisp" :build (:not compile)))
```

Why not flash words on the screen. Why not — hey, it could be fun.

```
(package! spray :pin "74d9dcfa2e8b38f96a43de9ab0eb13364300cb46")
```

With all our fancy Emacs themes, my terminal is missing out!

```
(package! theme-magic :pin "844c4311bd26ebafd4b6a1d72ddcc65d87f074e3")
```

What's even the point of using Emacs unless you're constantly telling everyone about it?

```
(package! elcord :pin "01b26d1af2f33a7c7c5a1c24d8bfb6d40115a7b0")
```

For some reason, I find myself demoing Emacs every now and then. Showing what keyboard stuff I'm doing on-screen seems helpful. While screenkey does exist, having something that doesn't cover up screen content is nice.


```
(package! keycast :pin "a3a0798349adf3e33277091fa8dee63173b68edf")
```

let's just make sure this is lazy-loaded appropriately.

```
(use-package! keycast
  :commands keycast-mode
  :config
  (define-minor-mode keycast-mode
```

```
    "Show current command and its key binding in the mode line."
    :global t
    (if keycast-mode
        (progn
          (add-hook 'pre-command-hook 'keycast--update t)
          (add-to-list 'global-mode-string '("" mode-line-keycast " ")))
      (remove-hook 'pre-command-hook 'keycast--update)
      (setq global-mode-string (remove '("" mode-line-keycast " ")
      ↪  global-mode-string))))
  (custom-set-faces!
    '(keycast-command :inherit doom-modeline-debug
                      :height 0.9)
    '(keycast-key :inherit custom-modified
                  :height 1.1
                  :weight bold)))
```

In a similar manner, gif-screencast may come in handy.

```
(package! gif-screencast :pin "1145e676b160e7b1e5756f5b0f30dd31de252e1f")
```

We can lazy load this using the start/stop commands.

I initially installed `scrot` for this, since it was the default capture program. However it raised `glib error: Saving to file ... failed` each time it was run. Google didn't reveal any easy fixed, so I switched to maim. We now need to pass it the window ID. This doesn't change throughout the lifetime of an emacs instance, so as long as a single window is used `xdotool getactivewindow` will give a satisfactory result.

It seems that when new colours appear, that tends to make `gifsicle` introduce artefacts. To avoid this we pre-populate the colour map using the current doom theme.

```
(use-package! gif-screencast
  :commands gif-screencast-mode
  :config
  (map! :map gif-screencast-mode-map
        :g "<f8>" #'gif-screencast-toggle-pause
        :g "<f9>" #'gif-screencast-stop)
  (setq gif-screencast-program "maim"
        gif-screencast-args `("--quality" "3" "-i" ,(string-trim-right
                                                     (shell-command-to-string
                                                      "xdotool getactivewindow")))
        gif-screencast-optimize-args '("--batch" "--optimize=3"
        ↪  "--usecolormap=/tmp/doom-color-theme"))
  (defun gif-screencast-write-colormap ()
    (f-write-text
     (replace-regexp-in-string
      "\n+" "\n"
      (mapconcat (lambda (c) (if (listp (cdr c))
```

```
                                (cadr c))) doom-themes--colors "\n"))
    'utf-8
    "/tmp/doom-color-theme" ))
  (gif-screencast-write-colormap)
  (add-hook 'doom-load-theme-hook #'gif-screencast-write-colormap))
```

3.2.3 Features

CalcTeX This is a nice extension to `calc`

```
(package! calctex :recipe (:host github :repo "johnbcoughlin/calctex"
                           :files ("*.el" "calctex/*.el" "calctex-contrib/*.el"
                           ↪  "org-calctex/*.el"))
  :pin "7fa2673c64e259e04aef684ccf09ef85570c388b")
```

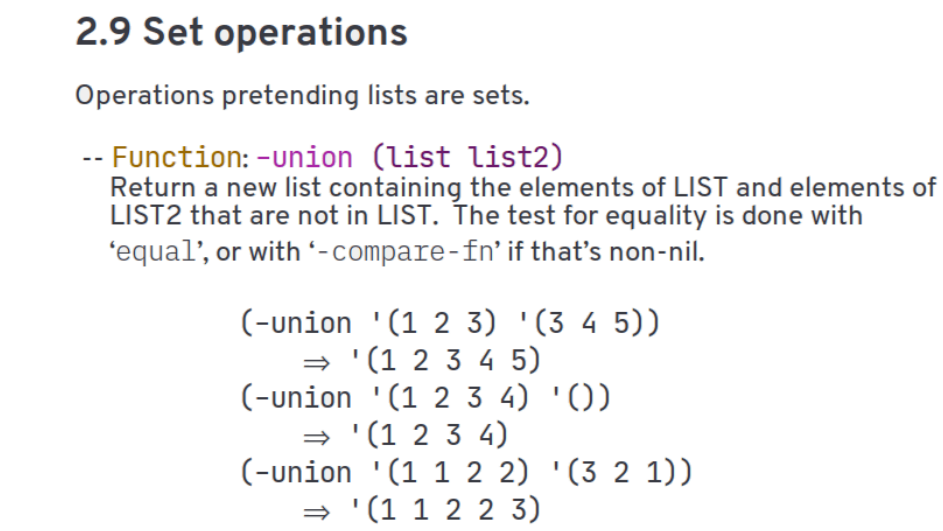
ESS View data frames better with

```
(package! ess-view :pin "d4e5a340b7bcc58c434867b97923094bd0680283")
```

Magit Delta Delta is a git diff syntax highlighter written in rust. The author also wrote a package to hook this into the magit diff view. This requires the `delta` binary.

```
;; (package! magit-delta :recipe (:host github :repo "dandavison/magit-delta") :pin
↪  "fc4de96e3faa1c983728239c5e41cc9f074b73a2")
```

Info colours This makes manual pages nicer to look at :) Variable pitch fontification + colouring




```
(package! info-colors :pin "47ee73cc19b1049eef32c9f3e264ea7ef2aaf8a5")
```

Large files The *very large files* mode loads large files in chunks, allowing one to open ridiculously large files.

```
(package! vlf :recipe (:host github :repo "m00natic/vlfi" :files ("*.el"))
  :pin "cc02f2533782d6b9b628cec7e2dcf25b2d05a27c" :disable t)
```

To make VLF available without delaying startup, we'll just load it in quiet moments.

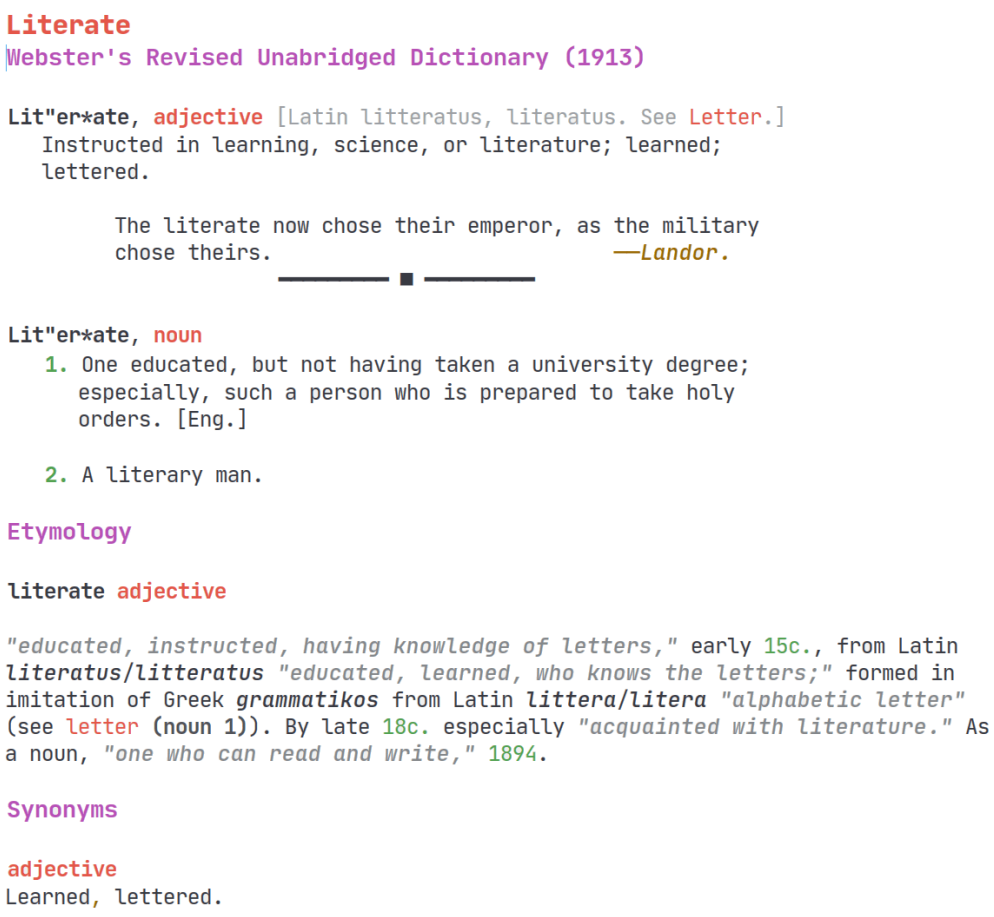
```
(use-package! vlf-setup
  :defer-incrementally vlf-tune vlf-base vlf-write vlf-search vlf-occur vlf-follow
  ↪  vlf-ediff vlf)
```

Definitions Doom already loads `define-word`, and provides it's own definition service using wordnut. However, using an offline dictionary possess a few compelling advantages, namely:

- speed
- integration of multiple dictionaries

GoldenDict seems like the best option currently avalible, but lacks a CLI. Hence, we'll fall back to sdcv (a CLI version of StarDict) for now. To interface with this, we'll use a my

`lexic` package.




```
(package! lexic :recipe (:local-repo "lisp/lexic"))
```

Given that a request for a CLI is the most upvoted issue on GitHub for GoldenDict, it's likely we'll be able to switch from `sdcv` to that in the future.

Since GoldenDict supports StarDict files, I expect this will be a relatively painless switch.

Calibre and ebook reading For managing my ebooks, I'll hook into the well-established ebook library manager calibre. A number of Emacs clients for this exist, but this seems like a good option.

```
(package! calibredb :pin "1f38fc34a8c159846450d18b1ee50cc960349ee7")
```

Then for reading them, the only currently viable options seems to be nov.el.

```
(package! nov :pin "0ece7ccbf79c074a3e4fbad1d1fa06647093f8e4")
```

Together these should give me a rather good experience reading ebooks.

Screenshots This makes it a breeze to take lovely screenshots.

```
(package! screenshot :recipe (:local-repo "lisp/screenshot"))
```

Some light configuring is all we need, so we can make use of the oxo wrapper file uploading script (which I've renamed to `upload`).

```
(use-package! screenshot
  :defer t
  :config (setq screenshot-upload-fn "upload %s 2>/dev/null"))
```

3.3 Language packages

3.3.1 LaTeX

For mathematical convenience, WIP

```
(package! auto-activating-snippets :recipe
  (:host github :repo "ymarco/auto-activating-snippets")
  :pin "85cd255347c4cc59e734eaec892599449ce48022")
(package! latex-auto-activating-snippets
  :recipe (:local-repo "lisp/LaTeX-auto-activating-snippets"))
```

And some basic config

```
(use-package! auto-activating-snippets
  :hook (LaTeX-mode . auto-activating-snippets-mode)
  :config (require 'latex-auto-activating-snippets))

(use-package! latex-auto-activating-snippets
  :config
  (defun als-tex-fold-maybe ()
    (unless (equal "/" als-transient-snippet-key)
      (+latex-fold-last-macro-a)))
  (add-hook 'aas-post-snippet-expand-hook #'als-tex-fold-maybe))
```

3.3.2 Org Mode

Use HEAD for development.

```
(unpin! org-mode)
```

Improve agenda/capture The agenda is nice, but a souped up version is nicer.

```
(package! org-super-agenda :pin "f5e80e4d0da6b2eeda9ba21e021838fa6a495376")
```

Similarly `doct` (Declarative Org Capture Templates) seems to be a nicer way to set up org-capture.

```
(package! doct
  :recipe (:host github :repo "progfolio/doct")
  :pin "8ac08633ae413a6605b6506d2739eece7475272e")
```

Visuals Org tables aren't the prettiest thing to look at. This package is supposed to redraw them in the buffer with box-drawing characters. Sounds like an improvement to me! Just need to get it working. . .

```
(package! org-pretty-table-mode
  :recipe (:host github :repo "Fuco1/org-pretty-table") :pin
  ↪  "474ad84a8fe5377d67ab7e491e8e68dac6e37a11")
```

For automatically toggling LaTeX fragment previews there's this nice package

```
(package! org-fragtog :pin "0151cabc7aa9f244f82e682b87713b344d780c23")
```

Then for pretty markers

```
(package! org-appear :recipe (:host github :repo "awth13/org-appear")
  :pin "19ea96e6e2ce01b8583b25a6e5579f1be207a119")
```

`org-superstar-mode` is great. While we're at it we may as well make tags prettier as well :)

```
(package! org-pretty-tags :pin "5c7521651b35ae9a7d3add4a66ae8cc176ae1c76")
```

There's this nice package that can provide nice syntax highlighting with LaTeX exports.

```
(package! engrave-faces :recipe (:local-repo "lisp/engrave-faces"))
```

```
(use-package! engrave-faces-latex
  :after ox-latex)
```

Extra functionality Because of the *lovely variety in markdown implementations* there isn't actually such a thing a standard table spec ... or standard anything really. Because `org-md` is a goody-two-shoes, it just uses HTML for all these non-standardised elements (a lot of them). So `ox-gfm` is handy for exporting markdown with all the features that GitHub has.

```
(package! ox-gfm :pin "99f93011b069e02b37c9660b8fcb45dab086a07f")
```

```
(use-package! ox-gfm
  :after org)
```

Now and then citations need to happen

```
(package! org-ref :pin "052a176b4cc1c080376c183e5e02ab37cb1f0f0a")
```

Came across this and ... it's cool

```
(package! org-graph-view :recipe (:host github :repo "alphapapa/org-graph-view")
↪  :pin "13314338d70d2c19511efccc491bed3ca0758170")
```

I **need** this in my life. It take a URL to a recipe from a common site, and inserts an org-ified version at point. Isn't that just great.

```
(package! org-chef :pin "5b461ed7d458cdcbff0af5013fbdbe88cbfb13a4")
```

Sometimes I'm given non-org files, that's very sad. Luckily Pandoc offers a way to make that right again, and this package makes that even easier to do.

```
(package! org-pandoc-import :recipe
  (:local-repo "lisp/org-pandoc-import" :files ("*.el" "filters" "preprocessors")))
```

```
(use-package! org-pandoc-import
  :after org)
```

Org-roam is nice by itself, but there are so *extra* nice packages which integrate with it.

```
(package! org-roam-server :pin "c7793202e9929dc2a415482779141e7b429421ce")
```

```
(use-package org-roam-server
  :after (org-roam server)
  :config
  (setq org-roam-server-host "127.0.0.1"
        org-roam-server-port 8078
        org-roam-server-export-inline-images t
        org-roam-server-authenticate nil
        org-roam-server-network-label-truncate t
        org-roam-server-network-label-truncate-length 60
        org-roam-server-network-label-wrap-length 20)
  (defun org-roam-server-open ()
    "Ensure the server is active, then open the roam graph."
    (interactive)
    (org-roam-server-mode 1)
    (browse-url-xdg-open (format "http://localhost:%d" org-roam-server-port))))
```

3.3.3 Systemd

For editing systemd unit files

```
(package! systemd :pin "51c148e09a129ddf33d95276aa0e89d4ef6f8dd2")
```

3.3.4 Graphviz

Graphviz is a nice method of visualising simple graphs, based on plaintext `.dot` / `.gv` files.

```
(package! graphviz-dot-mode :pin "3642a0a5f41a80c8ecef7c6143d514200b80e194")
```

3.3.5 Authinfo

```
(package! authinfo-color-mode
  :recipe (:local-repo "lisp/authinfo-color-mode"))
```

Now we just need to load it appropriately.

```
(use-package! authinfo-color-mode
  :mode ("authinfo.gpg\\'" . authinfo-color-mode)
  :init (advice-add 'authinfo-mode :override #'authinfo-color-mode))
```

3.3.6 Beancount (accounting)

```
(package! beancount :recipe (:host github :repo "beancount/beancount-mode")
  :pin "7a0ef01d1ff6f8c318af944131310ca06d4c65ff")
```

4 Package configuration

4.1 Abbrev mode

Thanks to use a single abbrev-table for multiple modes? - Emacs Stack Exchange I have the following.

```
(use-package abbrev
  :init
  (setq-default abbrev-mode t)
  ;; a hook funtion that sets the abbrev-table to org-mode-abbrev-table
  ;; whenever the major mode is a text mode
  (defun tec/set-text-mode-abbrev-table ()
```

```
    (if (derived-mode-p 'text-mode)
        (setq local-abbrev-table org-mode-abbrev-table)))
  :commands abbrev-mode
  :hook
  (abbrev-mode . tec/set-text-mode-abbrev-table)
  :config
  (setq abbrev-file-name (expand-file-name "abbrev.el" doom-private-dir))
  (setq save-abbrevs 'silently))
```

4.2 Calc

Radians are just better (setq calc-angle-mode 'rad ;; radians are rad calc-algebraic-mode t ;; allows '2*x instead of 'x<RET>2* calc-symbolic-mode t) ;; keeps stuff like ¿2 irrational for as long as possible (after! calctex (setq calctex-format-latex-header (concat calctex-format-latex-header "\arevmath")))

```
(add-hook 'calc-mode-hook #'calctex-mode)
```

4.3 Centaur Tabs

We want to make the tabs a nice, comfy size (36), with icons. The modifier marker is nice, but the particular default Unicode one causes a lag spike, so let's just switch to an o, which still looks decent but doesn't cause any issues. A 'active-bar' is nice, so let's have one of those. If we have it `under` needs us to turn on `x-underline-at-decent` though. For some reason this didn't seem to work inside the `(after! ... )` block ¿_(¿)_/¿. Then let's change the font to a sans serif, but the default one doesn't fit too well somehow, so let's switch to 'P22 Underground Book'; it looks much nicer.

```
(after! centaur-tabs
  (centaur-tabs-mode -1)
  (setq centaur-tabs-height 36
        centaur-tabs-set-icons t
        centaur-tabs-modified-marker "o"
        centaur-tabs-close-button "¿"
        centaur-tabs-set-bar 'above
        centaur-tabs-gray-out-icons 'buffer)
  (centaur-tabs-change-fonts "P22 Underground Book" 160))
;; (setq x-underline-at-descent-line t)
```

4.4 Company

It's nice to have completions almost all the time, in my opinion. Key strokes are just waiting to be saved!

```
(after! company
  (setq company-idle-delay 0.5
        company-minimum-prefix-length 2)
  (setq company-show-numbers t)
  (add-hook 'evil-normal-state-entry-hook #'company-abort)) ;; make aborting less
  ↪  annoying.
```

Now, the improvements from `precedent` are mostly from remembering history, so let's improve that memory.

```
(setq-default history-length 1000)
(setq-default prescient-history-length 1000)
```

4.4.1 Plain Text

`Ispell` is nice, let's have it in `text`, `markdown`, and `GFM`.

```
(set-company-backend!
  '(text-mode
    markdown-mode
    gfm-mode)
  '(:seperate
    company-ispell
    company-files
    company-yasnippet))
```

We then configure the dictionary we're using in Ispell.

4.4.2 ESS

`company-dabbrev-code` is nice. Let's have it.

```
(set-company-backend! 'ess-r-mode '(company-R-args company-R-objects
↪  company-dabbrev-code :separate))
```

4.5 Elcord

```
(setq elcord-use-major-mode-as-main-icon t)
```

4.6 Emacs Anywhere configuration

To start with, let's install this.

```
cd /tmp
curl -fsSL https://raw.github.com/zachcurry/emacs-anywhere/master/install -o
↪  ea-install.sh
sed -i
↪  's/EA_PATH=$HOME\/.emacs_anywhere/EA_PATH=$HOME\/.local\/share\/emacs_anywhere/'
↪  ea-install.sh
bash ea-install.sh || exit
cd ~/.local/share/emacs_anywhere
# Install in ~/.local not ~/.emacs_anywhere
sed -i 's/$HOME\/.emacs_anywhere/$HOME\/.local\/share\/emacs_anywhere/' ./bin/linux
↪  ./bin/emacstask
ln -s ~/.local/share/emacs_anywhere/bin/linux ~/.local/bin/emacs_anywhere
```

It's nice to recognise GitHub (so we can use `GFM`), and other apps which we know take markdown

```
(defun markdown-window-p (window-title)
  "Judges from WINDOW-TITLE whether the current window likes markdown"
  (if (string-match-p (rx (or "Stack Exchange" "Stack Overflow"
                              "Pull Request" "Issue" "Discord"))
                      window-title) t nil))
```

When the window opens, we generally want text so let's use a nice sans serif font, a position the window below and to the left. Oh, and don't forget about checking for `GFM`, otherwise let's just use `markdown`.

```
(defvar emacs-anywhere--active-markdown nil
  "Whether the buffer started off as markdown.
      Affects behaviour of `emacs-anywhere--finalise-content'")

(defun emacs-anywhere--finalise-content (&optional _frame)
  (when emacs-anywhere--active-markdown
    (fundamental-mode)
    (goto-char (point-min))
    (insert "#+property: header-args :exports both\n#+options: toc:nil\n")
    (rename-buffer "*EA Pre Export*")
```

```
    (org-export-to-buffer 'gfm ea--buffer-name)
    (kill-buffer "*EA Pre Export*"))
  (gui-select-text (buffer-string)))

(define-minor-mode emacs-anywhere-mode
  "To tweak the current buffer for some emacs-anywhere considerations"
  :init-value nil
  :keymap (list
           ;; Finish edit, but be smart in org mode
           (cons (kbd "C-c C-c")
                 (cmd! (if (and (eq major-mode 'org-mode)
                                (org-in-src-block-p))
                           (org-ctrl-c-ctrl-c)
                         (delete-frame))))
           ;; Abort edit. emacs-anywhere saves the current edit for next time.
           (cons (kbd "C-c C-k")
                 (cmd! (setq ea-on nil)
                       (delete-frame))))
  (when emacs-anywhere-mode
    ;; line breaking
    (turn-off-auto-fill)
    (visual-line-mode t)
    ;; DEL/C-SPC to clear (first keystroke only)
    (set-transient-map (let ((keymap (make-sparse-keymap)))
                         (define-key keymap (kbd "DEL")   (cmd! (delete-region
                         ↪  (point-min) (point-max))))
                         (define-key keymap (kbd "C-SPC") (cmd! (delete-region
                         ↪  (point-min) (point-max))))
                         keymap))
    ;; disable tabs
    (when (bound-and-true-p centaur-tabs-mode)
      (centaur-tabs-local-mode t))))

(defun ea-popup-handler (app-name window-title x y w h)
  (interactive)
  (set-frame-size (selected-frame) 80 12)
  ;; position the frame near the mouse
  (let* ((mousepos (split-string (shell-command-to-string "xdotool getmouselocation
  ↪  | sed -E \"s/ screen:0 window:[^ ]*|x:|y://g\""")))
         (mouse-x (- (string-to-number (nth 0 mousepos)) 100))
         (mouse-y (- (string-to-number (nth 1 mousepos)) 50)))
    (set-frame-position (selected-frame) mouse-x mouse-y))

  (set-frame-name (concat "Quick Edit ¿ " ea-app-name " ¿ "
                          (truncate-string-to-width
                           (string-trim
                            (string-trim-right window-title
                                               (format "-[A-Za-z0-9 ]*%s"
                                               ↪  ea-app-name))
                            "[\s-]+" "[\s-]+")
                           45 nil nil "¿")))

  (when-let ((selection (gui-get-selection 'PRIMARY)))
```

```
    (insert selection))

  ;; convert buffer to org mode if markdown
  (when (markdown-window-p window-title)
    (shell-command-on-region (point-min) (point-max)
                             "pandoc -f markdown -t org" nil t)
    (deactivate-mark) (goto-char (point-max)))

  ;; remove any blank newline at end
  (delete-trailing-whitespace)
  (delete-char (- (skip-chars-backward "\n")))

  ;; set major mode
  (org-mode)

  ;; set markdown status
  (setq-local emacs-anywhere--active-markdown (markdown-window-p window-title))

  (advice-add 'ea--delete-frame-handler :before #'emacs-anywhere--finalise-content)

  ;; I'll be honest with myself, I /need/ spellcheck
  (spell-fu-buffer)

  (evil-insert-state) ; start in insert
  (emacs-anywhere-mode 1))

(add-hook 'ea-popup-hook 'ea-popup-handler)
```

#+end$_{\text{src}}$

This new minor mode of ours will be nice for messages, so let's hook it in for Email and IRC.

```
(add-hook! '(mu4e-compose-mode org-msg-edit-mode circe-channel-mode)
↪  (emoticon-to-emoji 1))
```

4.7 Eros-eval

This makes the result of evals with `gr` and `gR` just slightly prettier. Every bit counts right?

```
(setq eros-eval-result-prefix "¿ ")
```

4.8 EVIL

I don't use `evil-escape-mode`, so I may as well turn it off, I've heard it contributes a typing delay. I'm not sure it's much, but it is an extra `pre-command-hook` that I don't benefit from, so...

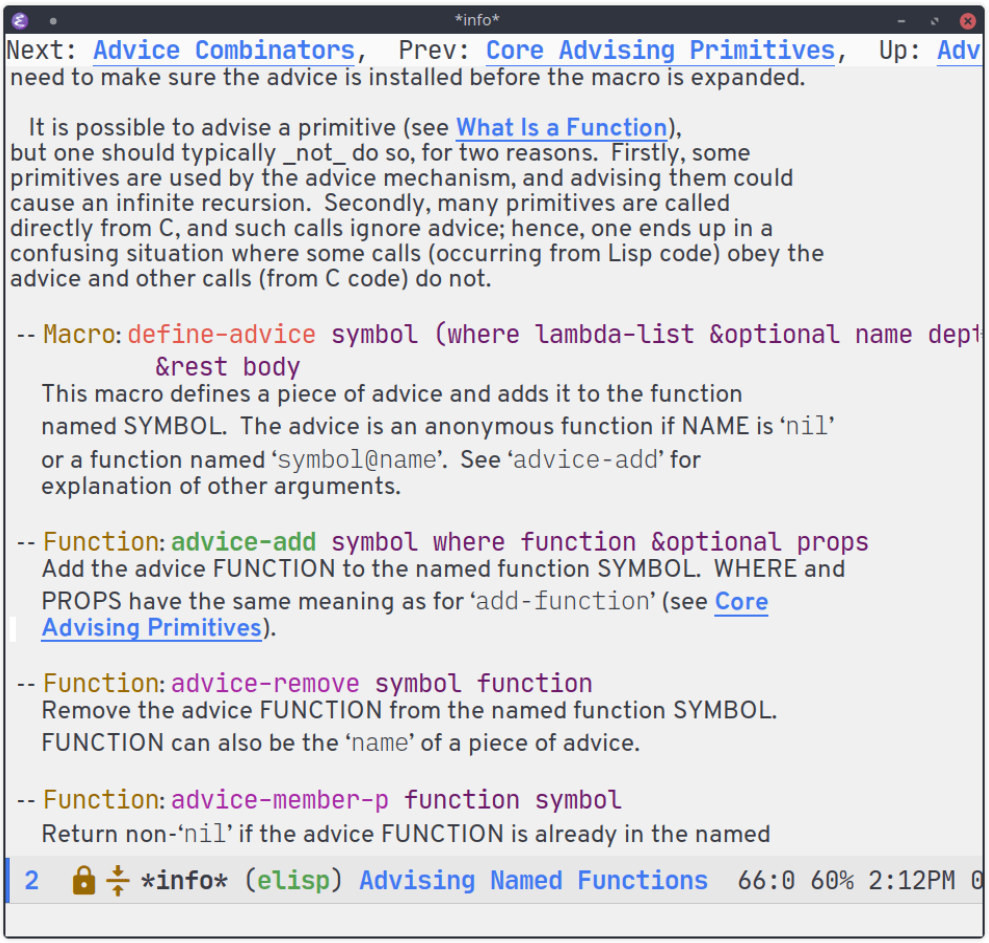
```
(after! evil-escape (evil-escape-mode -1))
```

When I want to make a substitution, I want it to be global more often than not — so let's make that the default.

```
(after! evil (setq evil-ex-substitute-global t)) ; I like my s/../.. to by global by
↪  default
```

4.9 Info colours

```
(use-package! info-colors
  :commands (info-colors-fontify-node))

(add-hook 'Info-selection-hook 'info-colors-fontify-node)

(add-hook 'Info-mode-hook #'mixed-pitch-mode)
```



4.10 Ispell

4.10.1 Downloading dictionaries

Let's get a nice big dictionary from SCOWL Custom List/Dictionary Creator with the following configuration

size 80 (huge)

spellings British(-ise) and Australian

spelling variants level 0

diacritics keep

extra lists hacker, roman numerals

Hunspell

```
cd /tmp
curl -o "hunspell-en-custom.zip"
↪  'http://app.aspell.net/create?max_size=80&spelling=GBs&spelling=AU&max_variant=0&diacritic=keep&spec
↪  numerals&encoding=utf-8&format=inline&download=hunspell'
unzip "hunspell-en-custom.zip"

sudo chown root:root en-custom.*
sudo mv en-custom.{aff,dic} /usr/share/myspell/
```

Aspell

```
cd /tmp
curl -o "aspell6-en-custom.tar.bz2"
↪  'http://app.aspell.net/create?max_size=80&spelling=GBs&spelling=AU&max_variant=0&diacritic=keep&spec
↪  numerals&encoding=utf-8&format=inline&download=aspell'
tar -xjf "aspell6-en-custom.tar.bz2"

cd aspell6-en-custom
./configure && make && sudo make install
```

4.10.2 Configuration

```
(setq ispell-dictionary "en-custom")
```

Oh, and by the way, if `company-ispell-dictionary` is `nil`, then `ispell-complete-word-dict` is used instead, which once again when `nil` is `ispell-alternate-dictionary`, which at the moment maps to a plaintext version of the above.

It seems reasonable to want to keep an eye on my personal dict, let's have it nearby (also means that if I change the 'main' dictionary I keep my addition).

```
(setq ispell-personal-dictionary (expand-file-name ".ispell_personal"
↪  doom-private-dir))
```

4.11 Ivy

While in an ivy mini-buffer `C-o` shows a list of all possible actions one may take. By default this is `#'ivy-read-action-by-key` however a better interface to this is using Hydra.

```
(setq ivy-read-action-function #'ivy-hydra-read-action)
```

I currently have ~40k functions. This seems like sufficient motivation to increase the maximum number of items ivy will sort to 40k + a bit, this way `SPC h f` et al. will continue to function as expected.

```
(setq ivy-sort-max-size 50000)
```

4.12 Magit

Magit is pretty nice by default. The diffs don't get any syntax-highlighting-love though which is a bit sad. Thankfully dandavison/magit-delta exists, which we can put to use.

```
;; (after! magit
;;   (magit-delta-mode +1))
```

Unfortunately this seems to mess things up, which is something I'll want to look into later.

4.13 Org Chef

Loading after org seems a bit premature. Let's just load it when we try to use it, either by command or in a capture template.

```
(use-package! org-chef
  :commands (org-chef-insert-recipe org-chef-get-recipe-from-url))
```

4.14 Projectile

Looking at documentation via `SPC h f` and `SPC h v` and looking at the source can add package src directories to projectile. This isn't desirable in my opinion.

```
(setq projectile-ignored-projects '("~/" "/tmp"
↪  "~/.emacs.d/.local/straight/repos/"))
(defun projectile-ignored-project-function (filepath)
  "Return t if FILEPATH is within any of `projectile-ignored-projects'"
  (or (mapcar (lambda (p) (s-starts-with-p p filepath))
  ↪  projectile-ignored-projects)))
```

4.15 Smart Parentheses

```
(sp-local-pair
 '(org-mode)
 "<<" ">>"
 :actions '(insert))
```

4.16 Spray

Let's make this suit me slightly better.

```
(setq spray-wpm 500
      spray-height 700)
```

4.17 Theme magic

Let's automatically update terminals on theme change (as long as `pywal` is available).

Unfortunately, as the theme is set on startup this causes the hook to be run immediately. It would be nicer to *not* have this add to our precious startup time (around 0.4s last time I checked). We can achieve this by deferring it with a short idle timer that should add the hook *just after* initialisation.

```
(run-with-idle-timer 0.1 nil (lambda () (add-hook 'doom-load-theme-hook
↪  'theme-magic-from-emacs)))
```

4.18 Tramp

Let's try to make tramp handle prompts better

```
(after! tramp
  (setenv "SHELL" "/bin/bash")
  (setq tramp-shell-prompt-pattern "\\(?:^\\|
      \\)[^]#$%>\n]*#?[]#$%>¿] *\\(\\[[0-9;]*[a-zA-Z] *\\)*")) ;; default + ¿
```

4.18.1 Troubleshooting

In case the remote shell is misbehaving, here are some things to try

Zsh There are some escape code you don't want, let's make it behave more considerately.

```
if [[ "$TERM" == "dumb" ]]; then
    unset zle_bracketed_paste
    unset zle
    PS1='$ '
    return
fi
```

4.18.2 Guix

Guix puts some binaries that TRAMP looks for in unexpected locations. That's no problem though, we just need to help TRAMP find them.

```
(after! tramp
  (appendq! tramp-remote-path
            '("~/.guix-profile/bin" "~/.guix-profile/sbin"
              "/run/current-system/profile/bin"
              "/run/current-system/profile/sbin")))
```

4.19 Treemacs

Quite often there are superfluous files I'm not that interested in. There's no good reason for them to take up space. Let's add a mechanism to ignore them.

```
(after! treemacs
  (defvar treemacs-file-ignore-extensions '()
    "File extension which `treemacs-ignore-filter' will ensure are ignored")
  (defvar treemacs-file-ignore-globs '()
    "Globs which will are transformed to `treemacs-file-ignore-regexps' which
    ↪  `treemacs-ignore-filter' will ensure are ignored")
  (defvar treemacs-file-ignore-regexps '()
    "RegExps to be tested to ignore files, generated from
    ↪  `treeemacs-file-ignore-globs'")
  (defun treemacs-file-ignore-generate-regexps ()
    "Generate `treemacs-file-ignore-regexps' from `treemacs-file-ignore-globs'"
    (setq treemacs-file-ignore-regexps (mapcar 'dired-glob-regexp
    ↪  treemacs-file-ignore-globs)))
  (if (equal treemacs-file-ignore-globs '()) nil
  ↪  (treemacs-file-ignore-generate-regexps))
  (defun treemacs-ignore-filter (file full-path)
    "Ignore files specified by `treemacs-file-ignore-extensions', and
    ↪  `treemacs-file-ignore-regexps'"
    (or (member (file-name-extension file) treemacs-file-ignore-extensions)
        (let ((ignore-file nil))
          (dolist (regexp treemacs-file-ignore-regexps ignore-file)
            (setq ignore-file (or ignore-file (if (string-match-p regexp full-path)
            ↪  t nil)))))))
  (add-to-list 'treemacs-ignored-file-predicates #'treemacs-ignore-filter))
```

Now, we just identify the files in question.

```
(setq treemacs-file-ignore-extensions
      '(;; LaTeX
        "aux"
        "ptc"
        "fdb_latexmk"
        "fls"
        "synctex.gz"
        "toc"
        ;; LaTeX - glossary
        "glg"
        "glo"
        "gls"
        "glsdefs"
        "ist"
        "acn"
        "acr"
        "alg"
        ;; LaTeX - pgfplots
        "mw"
        ;; LaTeX - pdfx
        "pdfa.xmpi"
        ))
(setq treemacs-file-ignore-globs
      '(;; LaTeX
```

```
        "*/_minted-*"
        ;; AucTeX
        "*/.auctex-auto"
        "*/_region_.log"
        "*/_region_.tex"))
```

4.20 Which-key

Let's make this popup a bit faster

```
(setq which-key-idle-delay 0.5) ;; I need the help, I really do
```

I also think that having `evil-` appear in so many popups is a bit too verbose, let's change that, and do a few other similar tweaks while we're at it.

```
(setq which-key-allow-multiple-replacements t)
(after! which-key
  (pushnew!
   which-key-replacement-alist
   '(("" . "\\`+?evil[-:]?\\(?:a-\\)?\\(.*\\)") . (nil . "¿\\1"))
   '(("\\`g s" . "\\`evilem--?motion-\\(.*\\)") . (nil . "¿\\1"))
   ))
```

```
SPC → lambda                          / → ◂avy-goto-char-timer
  # → ◂search-word-backward           a → ◂function-evil-forward-arg
  ( → ◂backward-sentence-begin        b → ◂backward-word-begin
  ) → ◂forward-sentence-begin         e → ◂forward-word-end
  * → ◂search-word-forward            f → ◂find-char
  + → ◂next-line-first-non-blank      j → ◂next-line
  - → ◂previous-line-first-non-bl..   k → ◂previous-line
g s-  (1 of 2) [C-h paging/help]
```

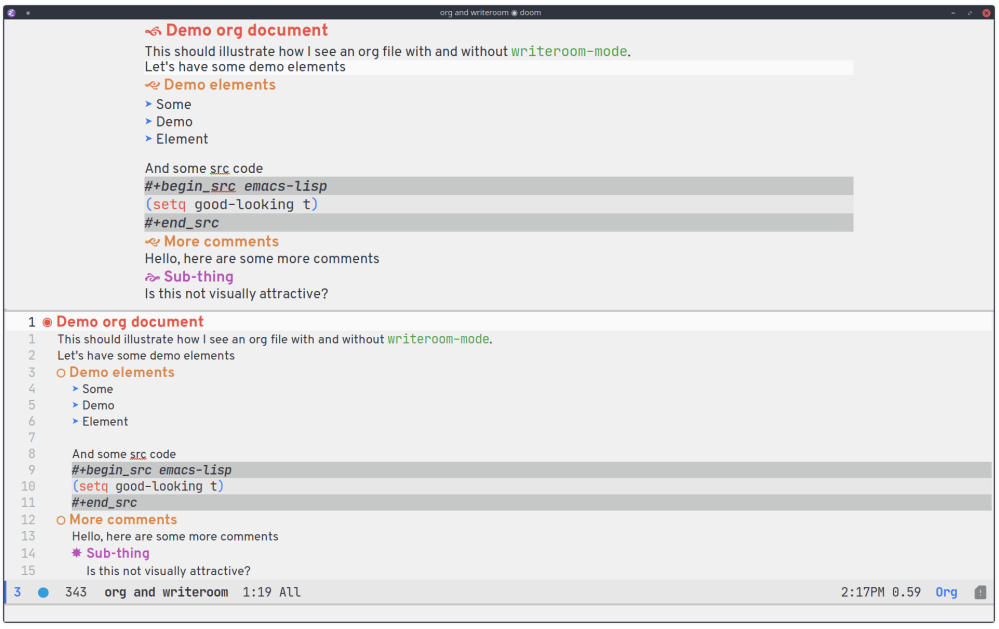
4.21 Writeroom

For starters, I think Doom is a bit over-zealous when zooming in

```
(setq +zen-text-scale 0.6)
```

Now, I think it would also be nice to remove line numbers and org stars in writeroom.

```
(after! writeroom-mode
  (add-hook 'writeroom-mode-hook
            (defun +zen-cleaner-org ()
              (when (and (eq major-mode 'org-mode) writeroom-mode)
                (setq-local -display-line-numbers display-line-numbers
                            display-line-numbers nil)
                (setq-local -org-indent-mode org-indent-mode)
                (org-indent-mode -1)
                (when (featurep 'org-superstar)
                  (setq-local -org-superstar-headline-bullets-list
                  ↪  org-superstar-headline-bullets-list
                             ;; org-superstar-headline-bullets-list '("¿" "¿" "¿"
                             ↪  "¿" "¿" "¿" "¿" "¿")
                             org-superstar-headline-bullets-list '("¿" "¿" "¿" "¿")
                             -org-superstar-remove-leading-stars
                             ↪  org-superstar-remove-leading-stars
                             org-superstar-remove-leading-stars t)
                  (org-superstar-restart)))))
  (add-hook 'writeroom-mode-disable-hook
            (defun +zen-dirty-org ()
              (when (eq major-mode 'org-mode)
                (setq-local display-line-numbers -display-line-numbers)
                (when -org-indent-mode
                  (org-indent-mode 1))
                (when (featurep 'org-superstar)
                  (setq-local org-superstar-headline-bullets-list
                  ↪ -org-superstar-headline-bullets-list
                             org-superstar-remove-leading-stars
                             ↪ -org-superstar-remove-leading-stars)
                  (org-superstar-restart))))))
```



4.22 xkcd

We want to set this up so it loads nicely in Extra links.

```
(use-package! xkcd
  :commands (xkcd-get-json
             xkcd-download xkcd-get
             ;; now for funcs from my extension of this pkg
             +xkcd-find-and-copy +xkcd-find-and-view
             +xkcd-fetch-info +xkcd-select)
  :config
  (setq xkcd-cache-dir (expand-file-name "xkcd/" doom-cache-dir)
        xkcd-cache-latest (concat xkcd-cache-dir "latest"))
  (unless (file-exists-p xkcd-cache-dir)
    (make-directory xkcd-cache-dir))
  (after! evil-snipe
    (add-to-list 'evil-snipe-disabled-modes 'xkcd-mode))
  :general (:states 'normal
            :keymaps 'xkcd-mode-map
            "<right>" #'xkcd-next
            "n"       #'xkcd-next ; evil-ish
            "<left>"  #'xkcd-prev
            "N"       #'xkcd-prev ; evil-ish
            "r"       #'xkcd-rand
            "a"       #'xkcd-rand ; because image-rotate can interfere
            "t"       #'xkcd-alt-text
            "q"       #'xkcd-kill-buffer
            "o"       #'xkcd-open-browser
            "e"       #'xkcd-open-explanation-browser
            ;; extras
            "s"       #'+xkcd-find-and-view
            "/"       #'+xkcd-find-and-view
            "y"       #'+xkcd-copy))
```

Let's also extend the functionality a whole bunch.

```
(after! xkcd
  (require 'emacsql-sqlite)

  (defun +xkcd-select ()
    "Prompt the user for an xkcd using `ivy-read' and `+xkcd-select-format'. Return
    ↪  the xkcd number or nil"
    (let* (prompt-lines
           (-dummy (maphash (lambda (key xkcd-info)
                              (push (+xkcd-select-format xkcd-info) prompt-lines))
                            +xkcd-stored-info))
           (num (ivy-read (format "xkcd (%s): " xkcd-latest) prompt-lines)))
      (if (equal "" num) xkcd-latest
        (string-to-number (replace-regexp-in-string "\\([0-9]+\\).*" "\\1" num)))))
```

```
(defun +xkcd-select-format (xkcd-info)
  "Creates each ivy-read line from an xkcd info plist. Must start with the xkcd
  ↪  number"
  (format "%-4s  %-30s %s"
          (propertize (number-to-string (plist-get xkcd-info :num))
                      'face 'counsel-key-binding)
          (plist-get xkcd-info :title)
          (propertize (plist-get xkcd-info :alt)
                      'face '(variable-pitch font-lock-comment-face))))

(defun +xkcd-fetch-info (&optional num)
  "Fetch the parsed json info for comic NUM. Fetches latest when omitted or 0"
  (require 'xkcd)
  (when (or (not num) (= num 0))
    (+xkcd-check-latest)
    (setq num xkcd-latest))
  (let ((res (or (gethash num +xkcd-stored-info)
                 (puthash num (+xkcd-db-read num) +xkcd-stored-info))))
    (unless res
      (+xkcd-db-write
       (let* ((url (format "https://xkcd.com/%d/info.0.json" num))
              (json-assoc
               (if (gethash num +xkcd-stored-info)
                   (gethash num +xkcd-stored-info)
                 (json-read-from-string (xkcd-get-json url num)))))
         json-assoc))
      (setq res (+xkcd-db-read num)))
    res))

;; since we've done this, we may as well go one little step further
(defun +xkcd-find-and-copy ()
  "Prompt for an xkcd using `+xkcd-select' and copy url to clipboard"
  (interactive)
  (+xkcd-copy (+xkcd-select)))

(defun +xkcd-copy (&optional num)
  "Copy a url to xkcd NUM to the clipboard"
  (interactive "i")
  (let ((num (or num xkcd-cur)))
    (gui-select-text (format "https://xkcd.com/%d" num))
    (message "xkcd.com/%d copied to clipboard" num)))

(defun +xkcd-find-and-view ()
  "Prompt for an xkcd using `+xkcd-select' and view it"
  (interactive)
  (xkcd-get (+xkcd-select))
  (switch-to-buffer "*xkcd*"))

(defvar +xkcd-latest-max-age (* 60 60) ; 1 hour
  "Time after which xkcd-latest should be refreshed, in seconds")

;; initialise `xkcd-latest' and `+xkcd-stored-info' with latest xkcd
(add-transient-hook! '+xkcd-select
```

```
  (require 'xkcd)
  (+xkcd-fetch-info xkcd-latest)
  (setq +xkcd-stored-info (+xkcd-db-read-all)))

(add-transient-hook! '+xkcd-fetch-info
  (xkcd-update-latest))

(defun +xkcd-check-latest ()
  "Use value in `xkcd-cache-latest' as long as it isn't older thabn
  ↪  `+xkcd-latest-max-age'"
  (unless (and (file-exists-p xkcd-cache-latest)
               (< (- (time-to-seconds (current-time))
                     (time-to-seconds (file-attribute-modification-time
                     ↪  (file-attributes xkcd-cache-latest))))
                  +xkcd-latest-max-age))
    (let* ((out (xkcd-get-json "http://xkcd.com/info.0.json" 0))
           (json-assoc (json-read-from-string out))
           (latest (cdr (assoc 'num json-assoc))))
      (when (/= xkcd-latest latest)
        (+xkcd-db-write json-assoc)
        (with-current-buffer (find-file xkcd-cache-latest)
          (setq xkcd-latest latest)
          (erase-buffer)
          (insert (number-to-string latest))
          (save-buffer)
          (kill-buffer (current-buffer)))))
    (shell-command (format "touch %s" xkcd-cache-latest))))

(defvar +xkcd-stored-info (make-hash-table :test 'eql)
  "Basic info on downloaded xkcds, in the form of a hashtable")

(defadvice! xkcd-get-json--and-cache (url &optional num)
  "Fetch the Json coming from URL.
      If the file NUM.json exists, use it instead.
      If NUM is 0, always download from URL.
      The return value is a string."
  :override #'xkcd-get-json
  (let* ((file (format "%s%d.json" xkcd-cache-dir num))
         (cached (and (file-exists-p file) (not (eq num 0))))
         (out (with-current-buffer (if cached
                                       (find-file file)
                                     (url-retrieve-synchronously url))
                (goto-char (point-min))
                (unless cached (re-search-forward "^$"))
                (prog1
                    (buffer-substring-no-properties (point) (point-max))
                  (kill-buffer (current-buffer))))))
    (unless (or cached (eq num 0))
      (xkcd-cache-json num out))
    out))

(defadvice! +xkcd-get (num)
  "Get the xkcd number NUM."
```

```
  :override 'xkcd-get
  (interactive "nEnter comic number: ")
  (xkcd-update-latest)
  (get-buffer-create "*xkcd*")
  (switch-to-buffer "*xkcd*")
  (xkcd-mode)
  (let (buffer-read-only)
    (erase-buffer)
    (setq xkcd-cur num)
    (let* ((xkcd-data (+xkcd-fetch-info num))
           (num (plist-get xkcd-data :num))
           (img (plist-get xkcd-data :img))
           (safe-title (plist-get xkcd-data :safe-title))
           (alt (plist-get xkcd-data :alt))
           title file)
      (message "Getting comic...")
      (setq file (xkcd-download img num))
      (setq title (format "%d: %s" num safe-title))
      (insert (propertize title
                          'face 'outline-1))
      (center-line)
      (insert "\n")
      (xkcd-insert-image file num)
      (if (eq xkcd-cur 0)
          (setq xkcd-cur num))
      (setq xkcd-alt alt)
      (message "%s" title))))

(defconst +xkcd-db--sqlite-available-p
  (with-demoted-errors "+org-xkcd initialization: %S"
    (emacsql-sqlite-ensure-binary)
    t))

(defvar +xkcd-db--connection (make-hash-table :test #'equal)
  "Database connection to +org-xkcd database.")

(defun +xkcd-db--get ()
  "Return the sqlite db file."
  (expand-file-name "xkcd.db" xkcd-cache-dir))

(defun +xkcd-db--get-connection ()
  "Return the database connection, if any."
  (gethash (file-truename xkcd-cache-dir)
           +xkcd-db--connection))

(defconst +xkcd-db--table-schema
  '((xkcds
     [(num integer :unique :primary-key)
      (year        :not-null)
      (month       :not-null)
      (link        :not-null)
      (news        :not-null)
      (safe_title  :not-null)
```

```
      (title       :not-null)
      (transcript  :not-null)
      (alt         :not-null)
      (img         :not-null)])))

(defun +xkcd-db--init (db)
  "Initialize database DB with the correct schema and user version."
  (emacsql-with-transaction db
    (pcase-dolist (`(,table . ,schema) +xkcd-db--table-schema)
      (emacsql db [:create-table $i1 $S2] table schema))))

(defun +xkcd-db ()
  "Entrypoint to the +org-xkcd sqlite database.
     Initializes and stores the database, and the database connection.
     Performs a database upgrade when required."
  (unless (and (+xkcd-db--get-connection)
               (emacsql-live-p (+xkcd-db--get-connection)))
    (let* ((db-file (+xkcd-db--get))
           (init-db (not (file-exists-p db-file))))
      (make-directory (file-name-directory db-file) t)
      (let ((conn (emacsql-sqlite db-file)))
        (set-process-query-on-exit-flag (emacsql-process conn) nil)
        (puthash (file-truename xkcd-cache-dir)
                 conn
                 +xkcd-db--connection)
        (when init-db
          (+xkcd-db--init conn)))))
  (+xkcd-db--get-connection))

(defun +xkcd-db-query (sql &rest args)
  "Run SQL query on +org-xkcd database with ARGS.
     SQL can be either the emacsql vector representation, or a string."
  (if  (stringp sql)
      (emacsql (+xkcd-db) (apply #'format sql args))
    (apply #'emacsql (+xkcd-db) sql args)))

(defun +xkcd-db-read (num)
  (when-let ((res
              (car (+xkcd-db-query [:select * :from xkcds
                                    :where (= num $s1)]
                                   num
                                   :limit 1))))
    (+xkcd-db-list-to-plist res)))

(defun +xkcd-db-read-all ()
  (let ((xkcd-table (make-hash-table :test 'eql :size 4000)))
    (mapcar (lambda (xkcd-info-list)
              (puthash (car xkcd-info-list) (+xkcd-db-list-to-plist
              ↪  xkcd-info-list) xkcd-table))
            (+xkcd-db-query [:select * :from xkcds]))
    xkcd-table))

(defun +xkcd-db-list-to-plist (xkcd-datalist)
```

```
    `(:num ,(nth 0 xkcd-datalist)
      :year ,(nth 1 xkcd-datalist)
      :month ,(nth 2 xkcd-datalist)
      :link ,(nth 3 xkcd-datalist)
      :news ,(nth 4 xkcd-datalist)
      :safe-title ,(nth 5 xkcd-datalist)
      :title ,(nth 6 xkcd-datalist)
      :transcript ,(nth 7 xkcd-datalist)
      :alt ,(nth 8 xkcd-datalist)
      :img ,(nth 9 xkcd-datalist)))

  (defun +xkcd-db-write (data)
    (+xkcd-db-query [:insert-into xkcds
                     :values $v1]
                    (list (vector
                           (cdr (assoc 'num        data))
                           (cdr (assoc 'year       data))
                           (cdr (assoc 'month      data))
                           (cdr (assoc 'link       data))
                           (cdr (assoc 'news       data))
                           (cdr (assoc 'safe_title data))
                           (cdr (assoc 'title      data))
                           (cdr (assoc 'transcript data))
                           (cdr (assoc 'alt        data))
                           (cdr (assoc 'img        data))
                           )))))
```

4.23 YASnippet

Nested snippets are good, enable that.

```
(setq yas-triggers-in-field t)
```

5 Applications

5.1 Ebooks

`calibredb` lets us use calibre through Emacs, because who wouldn't want to use something through Emacs?

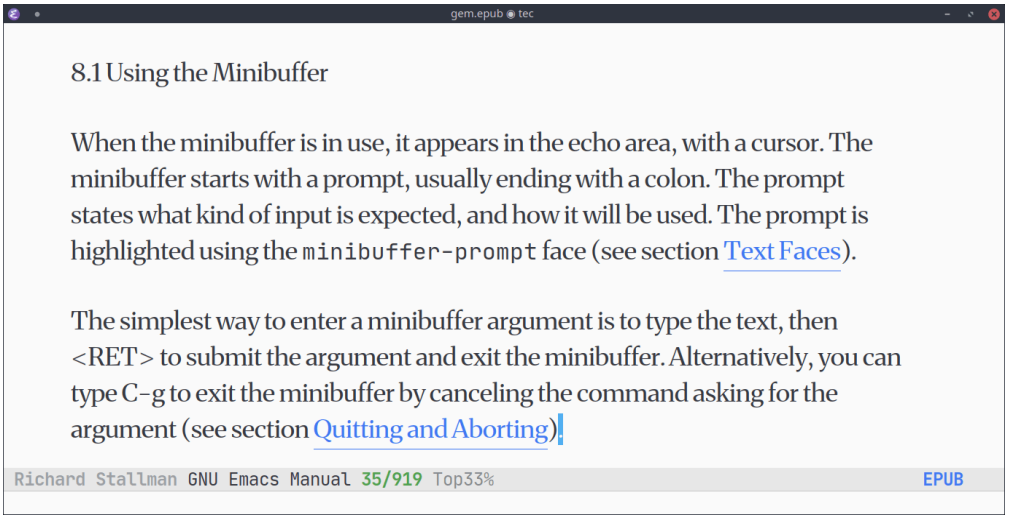
```
(use-package! calibredb
  :commands calibredb
```

Kindle I'm happy with my Kindle 2 so far, but if they cut off the free Wikipedia browsing, I plan to show up drunk on Jeff Bezos's lawn and refuse to leave.

```
:config
(setq calibredb-root-dir "~/Desktop/TEC/Other/Ebooks"
      calibredb-db-dir (expand-file-name "metadata.db" calibredb-root-dir))
(map! :map calibredb-show-mode-map
      :ne "?" #'calibredb-entry-dispatch
      :ne "o" #'calibredb-find-file
      :ne "O" #'calibredb-find-file-other-frame
      :ne "V" #'calibredb-open-file-with-default-tool
      :ne "s" #'calibredb-set-metadata-dispatch
      :ne "e" #'calibredb-export-dispatch
      :ne "q" #'calibredb-entry-quit
      :ne "." #'calibredb-open-dired
      :ne [tab] #'calibredb-toggle-view-at-point
      :ne "M-t" #'calibredb-set-metadata--tags
      :ne "M-a" #'calibredb-set-metadata--author_sort
      :ne "M-A" #'calibredb-set-metadata--authors
      :ne "M-T" #'calibredb-set-metadata--title
      :ne "M-c" #'calibredb-set-metadata--comments)
(map! :map calibredb-search-mode-map
      :ne [mouse-3] #'calibredb-search-mouse
      :ne "RET" #'calibredb-find-file
      :ne "?" #'calibredb-dispatch
      :ne "a" #'calibredb-add
      :ne "A" #'calibredb-add-dir
      :ne "c" #'calibredb-clone
      :ne "d" #'calibredb-remove
      :ne "D" #'calibredb-remove-marked-items
      :ne "j" #'calibredb-next-entry
      :ne "k" #'calibredb-previous-entry
      :ne "l" #'calibredb-virtual-library-list
      :ne "L" #'calibredb-library-list
      :ne "n" #'calibredb-virtual-library-next
      :ne "N" #'calibredb-library-next
      :ne "p" #'calibredb-virtual-library-previous
      :ne "P" #'calibredb-library-previous
      :ne "s" #'calibredb-set-metadata-dispatch
      :ne "S" #'calibredb-switch-library
      :ne "o" #'calibredb-find-file
      :ne "O" #'calibredb-find-file-other-frame
      :ne "v" #'calibredb-view
```

```
        :ne "V" #'calibredb-open-file-with-default-tool
        :ne "." #'calibredb-open-dired
        :ne "b" #'calibredb-catalog-bib-dispatch
        :ne "e" #'calibredb-export-dispatch
        :ne "r" #'calibredb-search-refresh-and-clear-filter
        :ne "R" #'calibredb-search-clear-filter
        :ne "q" #'calibredb-search-quit
        :ne "m" #'calibredb-mark-and-forward
        :ne "f" #'calibredb-toggle-favorite-at-point
        :ne "x" #'calibredb-toggle-archive-at-point
        :ne "h" #'calibredb-toggle-highlight-at-point
        :ne "u" #'calibredb-unmark-and-forward
        :ne "i" #'calibredb-edit-annotation
        :ne "DEL" #'calibredb-unmark-and-backward
        :ne [backtab] #'calibredb-toggle-view
        :ne [tab] #'calibredb-toggle-view-at-point
        :ne "M-n" #'calibredb-show-next-entry
        :ne "M-p" #'calibredb-show-previous-entry
        :ne "/" #'calibredb-search-live-filter
        :ne "M-t" #'calibredb-set-metadata--tags
        :ne "M-a" #'calibredb-set-metadata--author_sort
        :ne "M-A" #'calibredb-set-metadata--authors
        :ne "M-T" #'calibredb-set-metadata--title
        :ne "M-c" #'calibredb-set-metadata--comments))
```

Then, to actually read the ebooks we use nov.




```
(use-package! nov
  :mode ("\\.epub\\'" . nov-mode)
  :config
  (map! :map nov-mode-map
        :n "RET" #'nov-scroll-up)
```

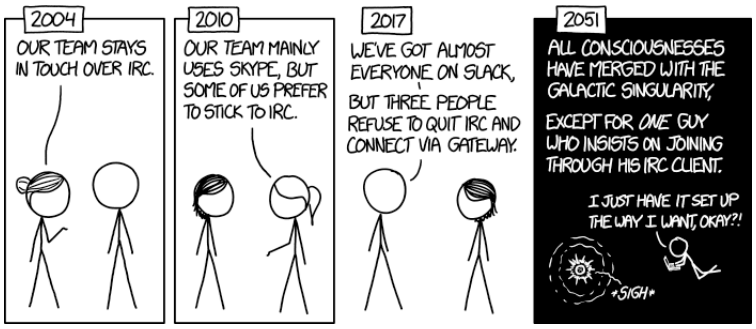
```
(defun doom-modeline-segment--nov-info ()
  (concat
   " "
   (propertize
    (cdr (assoc 'creator nov-metadata))
    'face 'doom-modeline-project-parent-dir)
   " "
   (cdr (assoc 'title nov-metadata))
   " "
   (propertize
    (format "%d/%d"
            (1+ nov-documents-index)
            (length nov-documents))
    'face 'doom-modeline-info)))

(advice-add 'nov-render-title :override #'ignore)

(defun +nov-mode-setup ()
  (face-remap-add-relative 'variable-pitch
                           :family "Merriweather"
                           :height 1.4
                           :width 'semi-expanded)
  (face-remap-add-relative 'default :height 1.3)
  (setq-local line-spacing 0.2
              next-screen-context-lines 4
              shr-use-colors nil)
  (require 'visual-fill-column nil t)
  (setq-local visual-fill-column-center-text t
              visual-fill-column-width 80
              nov-text-width 80)
  (visual-fill-column-mode 1)
  (hl-line-mode -1)

  (add-to-list '+lookup-definition-functions #'+lookup/dictionary-definition)

  (setq-local mode-line-format
              `((:eval
                 (doom-modeline-segment--workspace-name))
                (:eval
                 (doom-modeline-segment--window-number))
                (:eval
                 (doom-modeline-segment--nov-info))
                ,(propertize
                  " %P "
                  'face 'doom-modeline-buffer-minor-mode)
                ,(propertize
                  " "
                  'face (if (doom-modeline--active) 'mode-line
                  ↪  'mode-line-inactive)
                  'display `((space
                              :align-to
                              (- (+ right right-fringe right-margin)
```

```
                                    ,(* (let ((width (doom-modeline--font-width)))
                                          (or (and (= width 1) 1)
                                              (/ width (frame-char-width) 1.0)))
                                        (string-width
                                         (format-mode-line (cons "" '(:eval (doom-
                                         ↪  modeline-segment--major-mode)))))))))))
                        (:eval (doom-modeline-segment--major-mode)))))

(add-hook 'nov-mode-hook #'+nov-mode-setup))
```

5.2 IRC

`circe` is a client for IRC in Emacs (hey, isn't that a nice project name+acronym), and a greek enchantress who turned humans into animals.

Let's use the former to chat to ~~recluses~~ discerning individuals online.



Team Chat 2078: He announces that he's finally making the jump from screen+irssi to tmux+weechat.

Before we start seeing and sending messages, we need to authenticate with our IRC servers. The circe manual provided a snippet for putting some of the auth details in `.authinfo.gpg` — but I think we should go further than that: have the entire server info in our authinfo.

First, a reasonable format by which we can specify:

- server
- port
- SASL username
- SASL password

- channels to join

We can have these stored like so

```
machine chat.freenode.net login USERNAME password PASSWORD port PORT for irc
↪  channels emacs,org-mode
```

The `for irc` bit is used so we can uniquely identify all IRC auth info. By omitting the `#` in channel names we can have a list of channels comma-separated (no space!) which the secrets API will return as a single string.

```
(defun auth-server-pass (server)
  (if-let ((secret (plist-get (car (auth-source-search :host server)) :secret)))
      (if (functionp secret)
          (funcall secret) secret)
    (error "Could not fetch password for host %s" server)))

(defun register-irc-auths ()
  (require 'circe)
  (require 'dash)
  (let ((accounts (-filter (lambda (a) (string= "irc" (plist-get a :for)))
                           (auth-source-search :require '(:for) :max 10))))
    (appendq! circe-network-options
              (mapcar (lambda (entry)
                        (let* ((host (plist-get entry :host))
                               (label (or (plist-get entry :label) host))
                               (_ports (mapcar #'string-to-number
                                               (s-split "," (plist-get entry
                                               ↪  :port))))
                               (port (if (= 1 (length _ports)) (car _ports) _ports))
                               (user (plist-get entry :user))
                               (nick (or (plist-get entry :nick) user))
                               (channels (mapcar (lambda (c) (concat "#" c))
                                                 (s-split "," (plist-get entry
                                                 ↪  :channels)))))
                          `(,label
                            :host ,host :port ,port :nick ,nick
                            :sasl-username ,user :sasl-password auth-server-pass
                            :channels ,channels)))
                      accounts))))
```

We'll just call `(register-irc-auths)` on a hook when we start Circe up.

Now we're ready to go, let's actually wire-up Circe, with one or two configuration tweaks.

```
(after! circe
  (setq-default circe-use-tls t)
```

```
  (setq circe-notifications-alert-icon
  ↪  "/usr/share/icons/breeze/actions/24/network-connect.svg"
        lui-logging-directory "~/.emacs.d/.local/etc/irc"
        lui-logging-file-format "{buffer}/%Y/%m-%d.txt"
        circe-format-self-say "{nick:+13s} ¿ {body}")

  (custom-set-faces!
    '(circe-my-message-face :weight unspecified))

  (enable-lui-logging-globally)
  (enable-circe-display-images)

  <<org-emph-to-irc>>

  <<circe-emojis>>
  <<circe-emoji-alists>>

  (defun named-circe-prompt ()
    (lui-set-prompt
     (concat (propertize (format "%13s > " (circe-nick))
                         'face 'circe-prompt-face)
             "")))
  (add-hook 'circe-chat-mode-hook #'named-circe-prompt)

  (appendq! all-the-icons-mode-icon-alist
            '((circe-channel-mode all-the-icons-material "message" :face
            ↪  all-the-icons-lblue)
              (circe-server-mode all-the-icons-material "chat_bubble_outline" :face
              ↪  all-the-icons-purple))))

<<irc-authinfo-reader>>

(add-transient-hook! #'=irc (register-irc-auths))
```

5.2.1 Org-style emphasis

Let's do our **bold**, *italic*, and underline in org-syntax, using IRC control charachters

```
(defun lui-org-to-irc ()
  "Examine a buffer with simple org-mode formatting, and converts the empasis:
     *bold*, /italic/, and _underline_ to IRC semi-standard escape codes.
     =code= is converted to inverse (highlighted) text."
  (goto-char (point-min))
  (while (re-search-forward
  ↪  "\\_<\\(?1:[*/_=]\\)\\(?2:[^[:space:]]\\(?:.*?[^[:space:]]\\)?\\)\\1\\_>" nil
  ↪  t)
    (replace-match
     (concat (pcase (match-string 1)
               ("*" "")
```

```
                ("/" "")
                ("_" "")
                ("=" ""))
              (match-string 2)
              "") nil nil)))

(add-hook 'lui-pre-input-hook #'lui-org-to-irc)
```

5.2.2 Emojis

Let's setup Circe to use some emojis

```
(defun lui-ascii-to-emoji ()
  (goto-char (point-min))
  (while (re-search-forward "\\( \\)?::?\\([^[:space:]:]+\\):\\( \\)?" nil t)
    (replace-match
     (concat
      (match-string 1)
      (or (cdr (assoc (match-string 2) lui-emojis-alist))
          (concat ":" (match-string 2) ":"))
      (match-string 3))
     nil nil)))

(defun lui-emoticon-to-emoji ()
  (dolist (emoticon lui-emoticons-alist)
    (goto-char (point-min))
    (while (re-search-forward (concat " " (car emoticon) "\\( \\)?") nil t)
      (replace-match (concat " "
                             (cdr (assoc (cdr emoticon) lui-emojis-alist))
                             (match-string 1))))))

(define-minor-mode lui-emojify
  "Replace :emojis: and ;) emoticons with unicode emoji chars."
  :global t
  :init-value t
  (if lui-emojify
      (add-hook! lui-pre-input #'lui-ascii-to-emoji #'lui-emoticon-to-emoji)
    (remove-hook! lui-pre-input #'lui-ascii-to-emoji #'lui-emoticon-to-emoji)))
```

Now, some actual emojis to use.

```
(defvar lui-emojis-alist
  '(("grinning"                    . "¿")
    ("smiley"                      . "¿")
    ("smile"                       . "¿")
    ("grin"                        . "¿")
    ("laughing"                    . "¿")
```

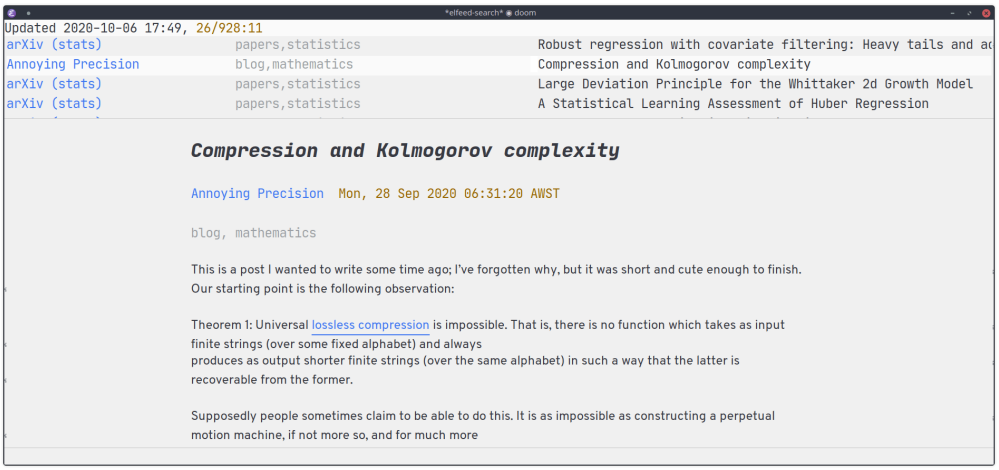
```
    ("sweat_smile"                    . "¿")
    ("joy"                            . "¿")
    ("rofl"                           . "¿")
    ("relaxed"                        . "¿¿")
    ("blush"                          . "¿")
    ("innocent"                       . "¿")
    ("slight_smile"                   . "¿")
    ("upside_down"                    . "¿")
    ("wink"                           . "¿")
    ("relieved"                       . "¿")
    ("heart_eyes"                     . "¿")
    ("yum"                            . "¿")
    ("stuck_out_tongue"               . "¿")
    ("stuck_out_tongue_closed_eyes"   . "¿")
    ("stuck_out_tongue_wink"          . "¿")
    ("zanzy"                          . "¿")
    ("raised_eyebrow"                 . "¿")
    ("monocle"                        . "¿")
    ("nerd"                           . "¿")
    ("cool"                           . "¿")
    ("star_struck"                    . "¿")
    ("party"                          . "¿")
    ("smirk"                          . "¿")
    ("unamused"                       . "¿")
    ("disapointed"                    . "¿")
    ("pensive"                        . "¿")
    ("worried"                        . "¿")
    ("confused"                       . "¿")
    ("slight_frown"                   . "¿")
    ("frown"                          . "¿¿")
    ("persevere"                      . "¿")
    ("confounded"                     . "¿")
    ("tired"                          . "¿")
    ("weary"                          . "¿")
    ("pleading"                       . "¿")
    ("tear"                           . "¿")
    ("cry"                            . "¿")
    ("sob"                            . "¿")
    ("triumph"                        . "¿")
    ("angry"                          . "¿")
    ("rage"                           . "¿")
    ("exploding_head"                 . "¿")
    ("flushed"                        . "¿")
    ("hot"                            . "¿")
    ("cold"                           . "¿")
    ("scream"                         . "¿")
    ("fearful"                        . "¿")
    ("disapointed"                    . "¿")
    ("relieved"                       . "¿")
    ("sweat"                          . "¿")
    ("thinking"                       . "¿")
    ("shush"                          . "¿")
    ("liar"                           . "¿")
```

```
    ("blank_face"                . "¿")
    ("neutral"                   . "¿")
    ("expressionless"            . "¿")
    ("grimace"                   . "¿")
    ("rolling_eyes"              . "¿")
    ("hushed"                    . "¿")
    ("frowning"                  . "¿")
    ("anguished"                 . "¿")
    ("wow"                       . "¿")
    ("astonished"                . "¿")
    ("sleeping"                  . "¿")
    ("drooling"                  . "¿")
    ("sleepy"                    . "¿")
    ("dizzy"                     . "¿")
    ("zipper_mouth"              . "¿")
    ("woozy"                     . "¿")
    ("sick"                      . "¿")
    ("vomiting"                  . "¿")
    ("sneeze"                    . "¿")
    ("mask"                      . "¿")
    ("bandaged_head"             . "¿")
    ("money_face"                . "¿")
    ("cowboy"                    . "¿")
    ("imp"                       . "¿")
    ("ghost"                     . "¿")
    ("alien"                     . "¿")
    ("robot"                     . "¿")
    ("clap"                      . "¿")
    ("thumpup"                   . "¿")
    ("+1"                        . "¿")
    ("thumbdown"                 . "¿")
    ("-1"                        . "¿")
    ("ok"                        . "¿")
    ("pinch"                     . "¿")
    ("left"                      . "¿")
    ("right"                     . "¿")
    ("down"                      . "¿")
    ("wave"                      . "¿")
    ("pray"                      . "¿")
    ("eyes"                      . "¿")
    ("brain"                     . "¿")
    ("facepalm"                  . "¿")
    ("tada"                      . "¿")
    ("fire"                      . "¿")
    ("flying_money"              . "¿")
    ("lighbulb"                  . "¿")
    ("heart"                     . "¿¿")
    ("sparkling_heart"           . "¿")
    ("heartbreak"                . "¿")
    ("100"                       . "¿")))

(defvar lui-emoticons-alist
  '((":)"    . "slight_smile")
```

```
    (";)"    . "wink")
    (":D"    . "smile")
    ("=D"    . "grin")
    ("xD"    . "laughing")
    (";("    . "joy")
    (":P"    . "stuck_out_tongue")
    (";D"    . "stuck_out_tongue_wink")
    ("xP"    . "stuck_out_tongue_closed_eyes")
    (":("    . "slight_frown")
    (";("    . "cry")
    (";'("   . "sob")
    (">:("   . "angry")
    (">>:(" . "rage")
    (":o"    . "wow")
    (":O"    . "astonished")
    (":/"    . "confused")
    (":-/"   . "thinking")
    (":|"    . "neutral")
    (":-|"   . "expressionless")))
```

5.3 Newsfeed

RSS feeds are still a thing. Why not make use of them with `elfeed`. I really like what fuxialexander has going on, but I don't think I need a custom module. Let's just try to patch on the main things I like the look of.



5.3.1 Keybindings

```
(map! :map elfeed-search-mode-map
      :after elfeed-search
      [remap kill-this-buffer] "q"
      [remap kill-buffer] "q"
      :n doom-leader-key nil
      :n "q" #'+rss/quit
      :n "e" #'elfeed-update
      :n "r" #'elfeed-search-untag-all-unread
      :n "u" #'elfeed-search-tag-all-unread
      :n "s" #'elfeed-search-live-filter
      :n "RET" #'elfeed-search-show-entry
      :n "p" #'elfeed-show-pdf
      :n "+" #'elfeed-search-tag-all
      :n "-" #'elfeed-search-untag-all
      :n "S" #'elfeed-search-set-filter
      :n "b" #'elfeed-search-browse-url
      :n "y" #'elfeed-search-yank)
(map! :map elfeed-show-mode-map
      :after elfeed-show
      [remap kill-this-buffer] "q"
      [remap kill-buffer] "q"
      :n doom-leader-key nil
      :nm "q" #'+rss/delete-pane
      :nm "o" #'ace-link-elfeed
      :nm "RET" #'org-ref-elfeed-add
      :nm "n" #'elfeed-show-next
      :nm "N" #'elfeed-show-prev
      :nm "p" #'elfeed-show-pdf
      :nm "+" #'elfeed-show-tag
      :nm "-" #'elfeed-show-untag
      :nm "s" #'elfeed-show-new-live-search
      :nm "y" #'elfeed-show-yank)
```

5.3.2 Usability enhancements

```
(after! elfeed-search
  (set-evil-initial-state! 'elfeed-search-mode 'normal))
(after! elfeed-show-mode
  (set-evil-initial-state! 'elfeed-show-mode   'normal))

(after! evil-snipe
  (push 'elfeed-show-mode   evil-snipe-disabled-modes)
  (push 'elfeed-search-mode evil-snipe-disabled-modes))
```

5.3.3 Visual enhancements

```
(after! elfeed

  (elfeed-org)
  (use-package! elfeed-link)

  (setq elfeed-search-filter "@1-week-ago +unread"
        elfeed-search-print-entry-function '+rss/elfeed-search-print-entry
        elfeed-search-title-min-width 80
        elfeed-show-entry-switch #'pop-to-buffer
        elfeed-show-entry-delete #'+rss/delete-pane
        elfeed-show-refresh-function #'+rss/elfeed-show-refresh--better-style
        shr-max-image-proportion 0.6)

  (add-hook! 'elfeed-show-mode-hook (hide-mode-line-mode 1))
  (add-hook! 'elfeed-search-update-hook #'hide-mode-line-mode)

  (defface elfeed-show-title-face '((t (:weight ultrabold :slant italic :height
  ↪  1.5)))
    "title face in elfeed show buffer"
    :group 'elfeed)
  (defface elfeed-show-author-face `((t (:weight light)))
    "title face in elfeed show buffer"
    :group 'elfeed)
  (set-face-attribute 'elfeed-search-title-face nil
                      :foreground 'nil
                      :weight 'light)

  (defadvice! +rss-elfeed-wrap-h-nicer ()
    "Enhances an elfeed entry's readability by wrapping it to a width of
       `fill-column' and centering it with `visual-fill-column-mode'."
    :override #'+rss-elfeed-wrap-h
    (let ((inhibit-read-only t)
          (inhibit-modification-hooks t))
      (setq-local truncate-lines nil)
      (setq-local shr-width 120)
      (setq-local line-spacing 0.2)
      (setq-local visual-fill-column-center-text t)
      (visual-fill-column-mode)
      ;; (setq-local shr-current-font '(:family "Merriweather" :height 1.2))
      (set-buffer-modified-p nil)))

  (defun +rss/elfeed-search-print-entry (entry)
    "Print ENTRY to the buffer."
    (let* ((elfeed-goodies/tag-column-width 40)
           (elfeed-goodies/feed-source-column-width 30)
           (title (or (elfeed-meta entry :title) (elfeed-entry-title entry) ""))
           (title-faces (elfeed-search--faces (elfeed-entry-tags entry)))
           (feed (elfeed-entry-feed entry))
           (feed-title
            (when feed
```

```
            (or (elfeed-meta feed :title) (elfeed-feed-title feed))))
          (tags (mapcar #'symbol-name (elfeed-entry-tags entry)))
          (tags-str (concat (mapconcat 'identity tags ",")))
          (title-width (- (window-width) elfeed-goodies/feed-source-column-width
                          elfeed-goodies/tag-column-width 4))

          (tag-column (elfeed-format-column
                       tags-str (elfeed-clamp (length tags-str)
                                              elfeed-goodies/tag-column-width
                                              elfeed-goodies/tag-column-width)
                       :left))
          (feed-column (elfeed-format-column
                        feed-title (elfeed-clamp
                        ↪  elfeed-goodies/feed-source-column-width
                                                 elfeed-goodies/feed-source-column-
                                                 ↪  width
                                                 elfeed-goodies/feed-source-column-
                                                 ↪  width)
                        :left)))

      (insert (propertize feed-column 'face 'elfeed-search-feed-face) " ")
      (insert (propertize tag-column 'face 'elfeed-search-tag-face) " ")
      (insert (propertize title 'face title-faces 'kbd-help title))
      (setq-local line-spacing 0.2)))

  (defun +rss/elfeed-show-refresh--better-style ()
    "Update the buffer to match the selected entry, using a mail-style."
    (interactive)
    (let* ((inhibit-read-only t)
           (title (elfeed-entry-title elfeed-show-entry))
           (date (seconds-to-time (elfeed-entry-date elfeed-show-entry)))
           (author (elfeed-meta elfeed-show-entry :author))
           (link (elfeed-entry-link elfeed-show-entry))
           (tags (elfeed-entry-tags elfeed-show-entry))
           (tagsstr (mapconcat #'symbol-name tags ", "))
           (nicedate (format-time-string "%a, %e %b %Y %T %Z" date))
           (content (elfeed-deref (elfeed-entry-content elfeed-show-entry)))
           (type (elfeed-entry-content-type elfeed-show-entry))
           (feed (elfeed-entry-feed elfeed-show-entry))
           (feed-title (elfeed-feed-title feed))
           (base (and feed (elfeed-compute-base (elfeed-feed-url feed)))))
      (erase-buffer)
      (insert "\n")
      (insert (format "%s\n\n" (propertize title 'face 'elfeed-show-title-face)))
      (insert (format "%s\t" (propertize feed-title 'face
      ↪  'elfeed-search-feed-face)))
      (when (and author elfeed-show-entry-author)
        (insert (format "%s\n" (propertize author 'face 'elfeed-show-author-face))))
      (insert (format "%s\n\n" (propertize nicedate 'face 'elfeed-log-date-face)))
      (when tags
        (insert (format "%s\n"
                        (propertize tagsstr 'face 'elfeed-search-tag-face))))
      ;; (insert (propertize "Link: " 'face 'message-header-name))
```

```
      ;; (elfeed-insert-link link link)
      ;; (insert "\n")
      (cl-loop for enclosure in (elfeed-entry-enclosures elfeed-show-entry)
               do (insert (propertize "Enclosure: " 'face 'message-header-name))
               do (elfeed-insert-link (car enclosure))
               do (insert "\n"))
      (insert "\n")
      (if content
          (if (eq type 'html)
              (elfeed-insert-html content base)
            (insert content))
        (insert (propertize "(empty)\n" 'face 'italic)))
      (goto-char (point-min))))

  )
```

5.3.4 Functionality enhancements

```
(after! elfeed-show
  (require 'url)

  (defvar elfeed-pdf-dir
    (expand-file-name "pdfs/"
                      (file-name-directory (directory-file-name
                      ↪  elfeed-enclosure-default-dir))))

  (defvar elfeed-link-pdfs
    '(("https://www.jstatsoft.org/index.php/jss/article/view/v0\\([^/]+\\)" .
    ↪  "https://www.jstatsoft.org/index.php/jss/article/view/v0\\1/v\\1.pdf")
      ("http://arxiv.org/abs/\\([^/]+\\)" . "https://arxiv.org/pdf/\\1.pdf"))
    "List of alists of the form (REGEX-FOR-LINK . FORM-FOR-PDF)")

  (defun elfeed-show-pdf (entry)
    (interactive
     (list (or elfeed-show-entry (elfeed-search-selected :ignore-region))))
    (let ((link (elfeed-entry-link entry))
          (feed-name (plist-get (elfeed-feed-meta (elfeed-entry-feed entry))
          ↪  :title))
          (title (elfeed-entry-title entry))
          (file-view-function
           (lambda (f)
             (when elfeed-show-entry
               (elfeed-kill-buffer))
             (pop-to-buffer (find-file-noselect f))))
          pdf)

      (let ((file (expand-file-name
                   (concat (subst-char-in-string ?/ ?, title) ".pdf")
                   (expand-file-name (subst-char-in-string ?/ ?, feed-name)
```

```
                                    elfeed-pdf-dir))))
        (if (file-exists-p file)
            (funcall file-view-function file)
          (dolist (link-pdf elfeed-link-pdfs)
            (when (and (string-match-p (car link-pdf) link)
                       (not pdf))
              (setq pdf (replace-regexp-in-string (car link-pdf) (cdr link-pdf)
              ↪  link))))
            (if (not pdf)
                (message "No associated PDF for entry")
              (message "Fetching %s" pdf)
              (unless (file-exists-p (file-name-directory file))
                (make-directory (file-name-directory file) t))
              (url-copy-file pdf file)
              (funcall file-view-function file))))))

  )
```

5.4 Dictionary

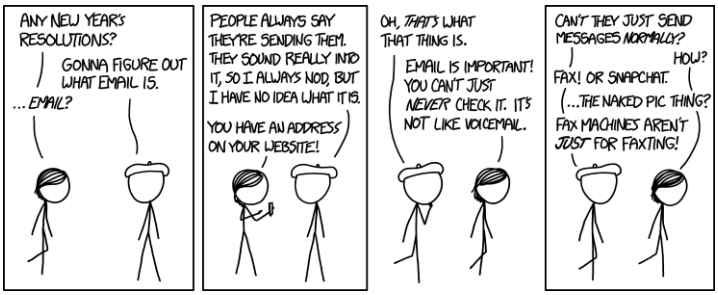
We start off by loading `lexic`, then we'll integrate it into pre-existing definition functionality (like `+lookup/dictionary-definition`).

```
(use-package! lexic
  :commands lexic-search lexic-list-dictionary
  :config
  (map! :map lexic-mode-map
        :n "q" #'lexic-return-from-lexic
        :nv "RET" #'lexic-search-word-at-point
        :n "a" #'outline-show-all
        :n "h" (cmd! (outline-hide-sublevels 3))
        :n "o" #'lexic-toggle-entry
        :n "n" #'lexic-next-entry
        :n "N" (cmd! (lexic-next-entry t))
        :n "p" #'lexic-previous-entry
        :n "P" (cmd! (lexic-previous-entry t))
        :n "E" (cmd! (lexic-return-from-lexic) ; expand
                     (switch-to-buffer (lexic-get-buffer)))
        :n "M" (cmd! (lexic-return-from-lexic) ; minimise
                     (lexic-goto-lexic))
        :n "C-p" #'lexic-search-history-backwards
        :n "C-n" #'lexic-search-history-forwards
        :n "/" (cmd! (call-interactively #'lexic-search))))
```

Now let's use this instead of wordnet.

```
(defadvice! +lookup/dictionary-definition-lexic (identifier &optional arg)
  "Look up the definition of the word at point (or selection) using
  ↪  `lexic-search'."
  :override #'+lookup/dictionary-definition
  (interactive
   (list (or (doom-thing-at-point-or-region 'word)
             (read-string "Look up in dictionary: "))
         current-prefix-arg))
  (lexic-search identifier nil nil t))
```

5.5 Mail



Email My New Year's resolution for 2014-54-12/30/14 Dec:12:1420001642 is to learn these stupid time formatting strings.

5.5.1 Fetching

The contenders for this seem to be:

- OfflineIMAP (ArchWiki page)
- isync/mbsync (ArchWiki page)

From perusing r/emacs the prevailing opinion seems to be that

- isync is faster
- isync works more reliably

So let's use that.

The config was straightforward, and is located at ~/.mbsyncrc. I'm currently successfully connecting to: Gmail, office365mail, and dovecot. I'm also shoving passwords in my

authinfo.gpg and fetching them using `PassCmd`:

```
gpg2 -q --for-your-eyes-only --no-tty -d ~/.authinfo.gpg | awk '/machine IMAP_SERCER
↪  login EMAIL_ADDR/ {print $NF}'
```

We can run `mbsync -a` in a systemd service file or something, but we can do better than that. vsemyonoff/easymail seems like the sort of thing we want, but is written for `notmuch` unfortunately. We can still use it for inspiration though. Using goimapnotify we should be able to sync just after new mail. Unfortunately this means *yet another* config file :(

We install with

```
go get -u gitlab.com/shackra/goimapnotify
ln -s ~/.local/share/go/bin/goimapnotify ~/.local/bin/
```

Here's the general plan:

1. Use `goimapnotify` to monitor mailboxes This needs it's own set of configs, and `systemd` services, which is a pain. We remove this pain by writing a python script (found below) to setup these config files, and systemd services by parsing the ~/.mbsyncrc file.

2. On new mail, call `mbsync --pull --new ACCOUNT:BOX` We try to be as specific as possible, so `mbsync` returns as soon as possible, and we can *get those emails as soon as possible*.

3. Try to call `mu index --lazy-fetch`. This fails if mu4e is already open (due to a write lock on the database), so in that case we just `touch` a tmp file (`/tmp/mu_reindex_now`).

4. Separately, we set up Emacs to check for the existance of `/tmp/mu_reindex_now` once a second while mu4e is running, and (after deleting the file) call `mu4e-update-index`.

Let's start off by handling the elisp side of things

Rebuild mail index while using mu4e

```
(after! mu4e
  (defvar mu4e-reindex-request-file "/tmp/mu_reindex_now"
    "Location of the reindex request, signaled by existance")
  (defvar mu4e-reindex-request-min-seperation 5.0
```

```
  "Don't refresh again until this many second have elapsed.
      Prevents a series of redisplays from being called (when set to an
  ↪  appropriate value)")

(defvar mu4e-reindex-request--file-watcher nil)
(defvar mu4e-reindex-request--file-just-deleted nil)
(defvar mu4e-reindex-request--last-time 0)

(defun mu4e-reindex-request--add-watcher ()
  (setq mu4e-reindex-request--file-just-deleted nil)
  (setq mu4e-reindex-request--file-watcher
        (file-notify-add-watch mu4e-reindex-request-file
                               '(change)
                               #'mu4e-file-reindex-request)))

(defadvice! mu4e-stop-watching-for-reindex-request ()
  :after #'mu4e~proc-kill
  (if mu4e-reindex-request--file-watcher
      (file-notify-rm-watch mu4e-reindex-request--file-watcher)))

(defadvice! mu4e-watch-for-reindex-request ()
  :after #'mu4e~proc-start
  (mu4e-stop-watching-for-reindex-request)
  (when (file-exists-p mu4e-reindex-request-file)
    (delete-file mu4e-reindex-request-file))
  (mu4e-reindex-request--add-watcher))

(defun mu4e-file-reindex-request (event)
  "Act based on the existance of `mu4e-reindex-request-file'"
  (if mu4e-reindex-request--file-just-deleted
      (mu4e-reindex-request--add-watcher)
    (when (equal (nth 1 event) 'created)
      (delete-file mu4e-reindex-request-file)
      (setq mu4e-reindex-request--file-just-deleted t)
      (mu4e-reindex-maybe t))))

(defun mu4e-reindex-maybe (&optional new-request)
  "Run `mu4e~proc-index' if it's been more than
      `mu4e-reindex-request-min-seperation'seconds since the last request,"
  (let ((time-since-last-request (- (float-time)
                                    mu4e-reindex-request--last-time)))
    (when new-request
      (setq mu4e-reindex-request--last-time (float-time)))
    (if (> time-since-last-request mu4e-reindex-request-min-seperation)
        (mu4e~proc-index nil t)
      (when new-request
        (run-at-time (* 1.1 mu4e-reindex-request-min-seperation) nil
                     #'mu4e-reindex-maybe))))))
```

Config transcoding & service management As long as the `mbsyncrc` file exists, this is as easy as running

`~/.config/doom/misc/mbsync-imapnotify.py`

When run without flags this will perform the following actions

- Read, and parse ~/.mbsyncrc, specifically recognising the following properties
  - `IMAPAccount`
  - `Host`
  - `Port`
  - `User`
  - `Password`
  - `PassCmd`
  - `Patterns`
- Call `mbsync --list ACCOUNT`, and filter results according to `Patterns`
- Construct a imapnotify config for each account, with the following hooks

  onNewMail `mbsync --pull ACCOUNT:MAILBOX`

  onNewMailPost `if mu index --lazy-check; then test -f /tmp/mu_reindex_now && rm /tmp/mu_reindex_now; else touch /tmp/mu_reindex_now; fi`
- Compare accounts list to previous accounts, enable/disable the relevant systemd services, called with the `--now` flag (start/stop services as well)

This script also supports the following flags

- `--status` to get the status of the relevant systemd services supports `active`, `failing`, and `disabled`
- `--enable` to enable all relevant systemd services
- `--disable` to disable all relevant systemd services

```
from pathlib import Path
import json
import re
import shutil
import subprocess
import sys
import fnmatch

mbsyncFile = Path("~/.mbsyncrc").expanduser()

imapnotifyConfigFolder = Path("~/.config/imapnotify/").expanduser()
imapnotifyConfigFolder.mkdir(exist_ok=True)
imapnotifyConfigFilename = "notify.conf"

imapnotifyDefault = {
    "host": "",
    "port": 993,
    "tls": True,
    "tlsOptions": {"rejectUnauthorized": True},
    "onNewMail": "",
    "onNewMailPost": "if mu index --lazy-check; then test -f /tmp/mu_reindex_now &&
    ↪  rm /tmp/mu_reindex_now; else touch /tmp/mu_reindex_now; fi",
}


def stripQuotes(string):
    if string[0] == '"' and string[-1] == '"':
        return string[1:-1].replace('\\"', '"')


mbsyncInotifyMapping = {
    "Host": (str, "host"),
    "Port": (int, "port"),
    "User": (str, "username"),
    "Password": (str, "password"),
    "PassCmd": (stripQuotes, "passwordCmd"),
    "Patterns": (str, "_patterns"),
}

oldAccounts = [d.name for d in imapnotifyConfigFolder.iterdir() if d.is_dir()]

currentAccount = ""
currentAccountData = {}

successfulAdditions = []


def processLine(line):
    newAcc = re.match(r"^IMAPAccount ([^#]+)", line)

    linecontent = re.sub(r"(^|[^\\])#.*", "", line).split(" ", 1)
    if len(linecontent) != 2:
        return
```

```
    parameter, value = linecontent

    if parameter == "IMAPAccount":
        if currentAccountNumber > 0:
            finaliseAccount()
        newAccount(value)
    elif parameter in mbsyncInotifyMapping.keys():
        parser, key = mbsyncInotifyMapping[parameter]
        currentAccountData[key] = parser(value)
    elif parameter == "Channel":
        currentAccountData["onNewMail"] = f"mbsync --pull --new {value}:'%s'"


def newAccount(name):
    global currentAccountNumber
    global currentAccount
    global currentAccountData
    currentAccountNumber += 1
    currentAccount = name
    currentAccountData = {}
    print(f"\n\033[1;32m{currentAccountNumber}\033[0;32m - {name}\033[0;37m")


def accountToFoldername(name):
    return re.sub(r"[^A-Za-z0-9]", "", name)


def finaliseAccount():
    if currentAccountNumber == 0:
        return

    global currentAccountData
    try:
        currentAccountData["boxes"] = getMailBoxes(currentAccount)
    except subprocess.CalledProcessError as e:
        print(
            f"\033[1;31mError:\033[0;31m failed to fetch mailboxes (skipping): "
            + f"`{' '.join(e.cmd)}' returned code {e.returncode}\033[0;37m"
        )
        return
    except subprocess.TimeoutExpired as e:
        print(
            f"\033[1;31mError:\033[0;31m failed to fetch mailboxes (skipping): "
            + f"`{' '.join(e.cmd)}' timed out after {e.timeout:.2f}
            ↪  seconds\033[0;37m"
        )
        return

    if "_patterns" in currentAccountData:
        currentAccountData["boxes"] = applyPatternFilter(
            currentAccountData["_patterns"], currentAccountData["boxes"]
        )
```

```
    # strip not-to-be-exported data
    currentAccountData = {
        k: currentAccountData[k] for k in currentAccountData if k[0] != "_"
    }

    parametersSet = currentAccountData.keys()
    currentAccountData = {**imapnotifyDefault, **currentAccountData}
    for key, val in currentAccountData.items():
        valColor = "\033[0;33m" if key in parametersSet else "\033[0;37m"
        print(f"  \033[1;37m{key:<13} {valColor}{val}\033[0;37m")

    if (
            len(currentAccountData["boxes"]) > 15
            and "@gmail.com" in currentAccountData["username"]
    ):
        print(
            "  \033[1;31mWarning:\033[0;31m Gmail raises an error when more than"
            + "\033[1;31m15\033[0;31m simultanious connections are attempted."
            + "\n            You are attempting to monitor "
            + f"\033[1;31m{len(currentAccountData['boxes'])}\033[0;31m
            ↪  mailboxes.\033[0;37m"
        )

    configFile = (
        imapnotifyConfigFolder
        / accountToFoldername(currentAccount)
        / imapnotifyConfigFilename
    )
    configFile.parent.mkdir(exist_ok=True)

    json.dump(currentAccountData, open(configFile, "w"), indent=2)
    print(f" \033[0;35mConfig generated and saved to {configFile}\033[0;37m")

    global successfulAdditions
    successfulAdditions.append(accountToFoldername(currentAccount))


def getMailBoxes(account):
    boxes = subprocess.run(
        ["mbsync", "--list", account], check=True, stdout=subprocess.PIPE,
        ↪  timeout=10.0
    )
    return boxes.stdout.decode("utf-8").strip().split("\n")


def applyPatternFilter(pattern, mailboxes):
    patternRegexs = getPatternRegexes(pattern)
    return [m for m in mailboxes if testPatternRegexs(patternRegexs, m)]


def getPatternRegexes(pattern):
    def addGlob(b):
```

```
        blobs.append(b.replace('\\"', '"'))
        return ""

    blobs = []
    pattern = re.sub(r' ?"([^"]+)"', lambda m: addGlob(m.groups()[0]), pattern)
    blobs.extend(pattern.split(" "))
    blobs = [
        (-1, fnmatch.translate(b[1::])) if b[0] == "!" else (1,
        ↪  fnmatch.translate(b))
        for b in blobs
    ]
    return blobs


def testPatternRegexs(regexCond, case):
    for factor, regex in regexCond:
        if factor * bool(re.match(regex, case)) < 0:
            return False
    return True


def processSystemdServices():
    keptAccounts = [acc for acc in successfulAdditions if acc in oldAccounts]
    freshAccounts = [acc for acc in successfulAdditions if acc not in oldAccounts]
    staleAccounts = [acc for acc in oldAccounts if acc not in successfulAdditions]

    if keptAccounts:
        print(f"\033[1;34m{len(keptAccounts)}\033[0;34m kept accounts:\033[0;37m")
        restartAccountSystemdServices(keptAccounts)

    if freshAccounts:
        print(f"\033[1;32m{len(freshAccounts)}\033[0;32m new accounts:\033[0;37m")
        enableAccountSystemdServices(freshAccounts)
    else:
        print(f"\033[0;32mNo new accounts.\033[0;37m")

    notActuallyEnabledAccounts = [
        acc for acc in successfulAdditions if not
        ↪  getAccountServiceState(acc)["enabled"]
    ]
    if notActuallyEnabledAccounts:
        print(
            f"\033[1;32m{len(notActuallyEnabledAccounts)}\033[0;32m accounts need
            ↪  re-enabling:\033[0;37m"
        )
        enableAccountSystemdServices(notActuallyEnabledAccounts)

    if staleAccounts:
        print(f"\033[1;33m{len(staleAccounts)}\033[0;33m removed
        ↪  accounts:\033[0;37m")
        disableAccountSystemdServices(staleAccounts)
    else:
        print(f"\033[0;33mNo removed accounts.\033[0;37m")
```

```
def enableAccountSystemdServices(accounts):
    for account in accounts:
        print(f" \033[0;32m - \033[1;37m{account:<18}", end="\033[0;37m",
        ↪  flush=True)
        if setSystemdServiceState(
                "enable", f"goimapnotify@{accountToFoldername(account)}.service"
        ):
            print("\033[1;32m enabled")


def disableAccountSystemdServices(accounts):
    for account in accounts:
        print(f" \033[0;33m - \033[1;37m{account:<18}", end="\033[0;37m",
        ↪  flush=True)
        if setSystemdServiceState(
                "disable", f"goimapnotify@{accountToFoldername(account)}.service"
        ):
            print("\033[1;33m disabled")


def restartAccountSystemdServices(accounts):
    for account in accounts:
        print(f" \033[0;34m - \033[1;37m{account:<18}", end="\033[0;37m",
        ↪  flush=True)
        if setSystemdServiceState(
                "restart", f"goimapnotify@{accountToFoldername(account)}.service"
        ):
            print("\033[1;34m restarted")


def setSystemdServiceState(state, service):
    try:
        enabler = subprocess.run(
            ["systemctl", "--user", state, service, "--now"],
            check=True,
            stderr=subprocess.DEVNULL,
            timeout=5.0,
        )
        return True
    except subprocess.CalledProcessError as e:
        print(
            f" \033[1;31mfailed\033[0;31m to {state}, `{' '.join(e.cmd)}'"
            + f"returned code {e.returncode}\033[0;37m"
        )
    except subprocess.TimeoutExpired as e:
        print(f" \033[1;31mtimed out after {e.timeout:.2f} seconds\033[0;37m")
        return False


def getAccountServiceState(account):
    return {
```

```
        state: bool(
            1
            - subprocess.run(
                [
                    "systemctl",
                    "--user",
                    f"is-{state}",
                    "--quiet",
                    f"goimapnotify@{accountToFoldername(account)}.service",
                ],
                stderr=subprocess.DEVNULL,
            ).returncode
        )
        for state in ("enabled", "active", "failing")
    }


def getAccountServiceStates(accounts):
    for account in accounts:
        enabled, active, failing = getAccountServiceState(account).values()
        print(f"  - \033[1;37m{account:<18}\033[0;37m ", end="", flush=True)
        if not enabled:
            print("\033[1;33mdisabled\033[0;37m")
        elif active:
            print("\033[1;32mactive\033[0;37m")
        elif failing:
            print("\033[1;31mfailing\033[0;37m")
        else:
            print("\033[1;35min an unrecognised state\033[0;37m")


if len(sys.argv) > 1:
    if sys.argv[1]   in ["-e", "--enable"]:
        enableAccountSystemdServices(oldAccounts)
        exit()
    elif sys.argv[1] in ["-d", "--disable"]:
        disableAccountSystemdServices(oldAccounts)
        exit()
    elif sys.argv[1] in ["-r", "--restart"]:
        restartAccountSystemdServices(oldAccounts)
        exit()
    elif sys.argv[1] in ["-s", "--status"]:
        getAccountServiceStates(oldAccounts)
        exit()
    elif sys.argv[1] in ["-h", "--help"]:
```

```
        print("""\033[1;37mMbsync to IMAP Notify config generator.\033[0;37m
            Usage: mbsync-imapnotify [options]
            Options:
                -e, --enable       enable all services
                -d, --disable      disable all services
                -r, --restart      restart all services
                -s, --status       fetch the status for all services
                -h, --help         show this help
            """, end='')
        exit()
    else:
        print(f"\033[0;31mFlag {sys.argv[1]} not recognised, try --help\033[0;37m")
        exit()


mbsyncData = open(mbsyncFile, "r").read()

currentAccountNumber = 0

totalAccounts = len(re.findall(r"^IMAPAccount", mbsyncData, re.M))


def main():
    print("\033[1;34m:: MbSync to Go IMAP notify config file creator ::\033[0;37m")

    shutil.rmtree(imapnotifyConfigFolder)
    imapnotifyConfigFolder.mkdir(exist_ok=False)
    print("\033[1;30mImap Notify config dir purged\033[0;37m")

    print(f"Identified \033[1;32m{totalAccounts}\033[0;32m accounts.\033[0;37m")

    for line in mbsyncData.split("\n"):
        processLine(line)

    finaliseAccount()

    print(
        f"\nConfig files generated for
        ↪  \033[1;36m{len(successfulAdditions)}\033[0;36m"
        + f" out of \033[1;36m{totalAccounts}\033[0;37m accounts.\n"
    )

    processSystemdServices()


if __name__ == "__main__":
    main()
```

Systemd We then have a service file to run `goimapnotify` on all of these generated config files. We'll use a template service file so we can enable a unit per-account.

```
[Unit]
Description=IMAP notifier using IDLE, golang version.
ConditionPathExists=%h/.config/imapnotify/%I/notify.conf
After=network.target

[Service]
ExecStart=%h/.local/bin/goimapnotify -conf %h/.config/imapnotify/%I/notify.conf
Restart=always
RestartSec=30

[Install]
WantedBy=default.target
```

Enabling the service is actually taken care of by that python script.

From one or two small tests, this can bring the delay down to as low as five seconds, which I'm quite happy with.

This works well for fetching new mail, but we also want to propagate other changes (e.g. marking mail as read), and make sure we're up to date at the start, so for that I'll do the 'normal' thing and run `mbsync -all` every so often — let's say five minutes.

We can accomplish this via a systemd timer, and service file.

```
[Unit]
Description=call mbsync on all accounts every 5 minutes
ConditionPathExists=%h/.mbsyncrc

[Timer]
OnBootSec=5m
OnUnitInactiveSec=5m

[Install]
WantedBy=default.target
```

```
[Unit]
Description=mbsync service, sync all mail
Documentation=man:mbsync(1)
ConditionPathExists=%h/.mbsyncrc

[Service]
Type=oneshot
ExecStart=/usr/bin/mbsync -c %h/.mbsyncrc --all

[Install]
WantedBy=mail.target
```

Enabling (and starting) this is as simple as

```
systemctl --user enable mbsync.timer --now
```

5.5.2 Indexing/Searching

This is performed by Mu. This is a tool for finding emails stored in the Maildir format. According to the homepage, it's main features are

- Fast indexing
- Good searching
- Support for encrypted and signed messages
- Rich CLI tooling
- accent/case normalisation
- strong integration with email clients

Unfortunately `mu` is not currently packaged from me. Oh well, I guess I'm building it from source then. I needed to install these packages

- `gmime-devel`
- `xapian-core-devel`

```
cd ~/.local/lib/
git clone https://github.com/djcb/mu.git
cd ./mu
./autogen.sh
make
sudo make install
```

To check how my version compares to the latest published:

```
curl --silent "https://api.github.com/repos/djcb/mu/releases/latest" | grep
↪  '"tag_name":' | sed -E 's/.*"([^"]+)".*/\1/'
mu --version | head -n 1 | sed 's/.* version //'
```

5.5.3 Sending

SmtpMail seems to be the 'default' starting point, but that's not packaged for me. msmtp is however, so I'll give that a shot. Reading around a bit (googling "msmtp vs sendmail" for example) almost every comparison mentioned seems to suggest msmtp to be a better choice. I have seen the following points raised

- `sendmail` has several vulnerabilities
- `sendmail` is tedious to configure
- `ssmtp` is no longer maintained
- `msmtp` is a maintained alternative to `ssmtp`
- `msmtp` is easier to configure

The config file is ~/.msmtprc

System hackery Unfortunately, I seem to have run into a bug present in my packaged version, so we'll just install the latest from source.

For full use of the `auth` options, I need `GNU SASL`, which isn't packaged for me. I don't think I want it, but in case I do, I'll need to do this.

```
export GSASL_VERSION=1.8.1
cd ~/.local/lib/
curl "ftp://ftp.gnu.org/gnu/gsasl/libgsasl-$GSASL_VERSION.tar.gz" | tar xz
curl "ftp://ftp.gnu.org/gnu/gsasl/gsasl-$GSASL_VERSION.tar.gz" | tar xz
cd "./libgsasl-$GSASL_VERSION"
./configure
make
sudo make install
cd ..
cd "./gsasl-$VERSION"
./configure
make
sudo make install
cd ..
```

Now actually compile `msmtp`.

```
cd ~/.local/lib/
git clone https://github.com/marlam/msmtp-mirror.git ./msmtp
cd ./msmtp
libtoolize --force
aclocal
autoheader
automake --force-missing --add-missing
autoconf
# if using GSASL
# PKG_CONFIG_PATH=/usr/local/lib/pkgconfig ./configure --with-libgsasl
./configure
make
sudo make install
```

If using GSASL (from earlier) we need to make ensure that the dynamic library in in the library path. We can do by adding an executable with the same name earlier on in my $PATH.

```
LD_LIBRARY_PATH=/usr/local/lib exec /usr/local/bin/msmtp "$@"
```

5.5.4 Mu4e

Webmail clients are nice and all, but I still don't believe that SPAs in my browser can replaced desktop apps . . . sorry Gmail. I'm also liking google less and less.

Mailspring is a decent desktop client, quite lightweight for electron (apparently the backend is in C, which probably helps), however I miss Emacs stuff.

While Notmuch seems very promising, and I've heard good things about it, it doesn't seem to make any changes to the emails themselves. All data is stored in Notmuch's database. While this is a very interesting model, occasionally I need to pull up an email on say my phone, and so not I want the tagging/folders etc. to be applied to the mail itself — not stored in a database.

On the other hand Mu4e is also talked about a lot in positive terms, and seems to possess a similarly strong feature set — and modifies the mail itself (I.e. information is accessible without the database). Mu4e also seems to have a large user base, which tends to correlate with better support and attention.

As I installed mu4e from source, I need to add the /usr/local/ loadpath so Mu4e has a chance of loading

```
(add-to-list 'load-path "/usr/local/share/emacs/site-lisp/mu4e")
```

Viewing Mail There seem to be some advantages with using Gnus' article view (such as inline images), and judging from djcb/mu!1442 (comment) this seems to be the 'way of the future' for mu4e.

There are some all-the-icons font related issues, so we need to redefine the fancy chars, and make sure they get the correct width.

To account for the increase width of each flag character, and make perform a few more visual tweaks, we'll tweak the headers a bit

```
(after! mu4e
  (setq mu4e-headers-fields
        '((:flags . 6)
          (:account-stripe . 2)
          (:from-or-to . 25)
          (:folder . 10)
          (:recipnum . 2)
          (:subject . 80)
          (:human-date . 8))
        +mu4e-min-header-frame-width 142
        mu4e-headers-date-format "%d/%m/%y"
        mu4e-headers-time-format "¿ %H:%M"
        mu4e-headers-results-limit 1000
        mu4e-index-cleanup t)

  (add-to-list 'mu4e-bookmarks
               '(:name "Yesterday's messages" :query "date:2d..1d" :key ?y) t)

  (defvar +mu4e-header--folder-colors nil)
  (appendq! mu4e-header-info-custom
            '((:folder .
               (:name "Folder" :shortname "Folder" :help "Lowest level folder"
               ↪  :function
                (lambda (msg)
                  (+mu4e-colorize-str
                   (replace-regexp-in-string "\\`.*/" "" (mu4e-message-field msg
                   ↪  :maildir))
                   '+mu4e-header--folder-colors))))))
```

We'll also use a nicer alert icon

```
(setq mu4e-alert-icon "/usr/share/icons/Papirus/64x64/apps/evolution.svg")
```

Sending Mail Let's send emails too.

```
(after! mu4e
  (setq sendmail-program "/usr/bin/msmtp"
        send-mail-function #'smtpmail-send-it
        message-sendmail-f-is-evil t
        message-sendmail-extra-arguments '("--read-envelope-from"); ,
        ↪  "--read-recipients")
        message-send-mail-function #'message-send-mail-with-sendmail))
```

It's also nice to avoid accidentally sending emails with the wrong account. If we can send from the address in the `To` field, let's do that. Opening an `ivy` prompt otherwise also seems sensible.

We can register Emacs as a potential email client with the following desktop file, thanks to Etienne Deparis's Mu4e customization.

```
[Desktop Entry]
Name=Compose message in Emacs
GenericName=Compose a new message with Mu4e in Emacs
Comment=Open mu4e compose window
MimeType=x-scheme-handler/mailto;
Exec=emacsclient -create-frame --alternate-editor="" --no-wait --eval '(progn
↪  (x-focus-frame nil) (mu4e-compose-from-mailto "%u"))'
Icon=emacs
Type=Application
Terminal=false
Categories=Network;Email;
StartupWMClass=Emacs
```

To register this, just call

```
update-desktop-database ~/.local/share/applications
```

We also want to define `mu4e-compose-from-mailto`.

```
(defun mu4e-compose-from-mailto (mailto-string)
  (require 'mu4e)
  (unless mu4e~server-props (mu4e t) (sleep-for 0.1))
  (let* ((mailto (rfc2368-parse-mailto-url mailto-string))
         (to (cdr (assoc "To" mailto)))
         (subject (or (cdr (assoc "Subject" mailto)) ""))
         (body (cdr (assoc "Body" mailto)))
         (org-msg-greeting-fmt (if (assoc "Body" mailto)
                                   (replace-regexp-in-string "%" "%%"
                                                             (cdr (assoc "Body"
                                                             ↪  mailto)))
                                 org-msg-greeting-fmt))
```

```
        (headers (-filter (lambda (spec) (not (-contains-p '("To" "Subject" "Body")
        ↪  (car spec)))) mailto)))
    (mu4e~compose-mail to subject headers)))
```

This may not quite function as intended for now due to jeremy-compostella/org-msg#52.

5.5.5 Org Msg

Doom does a fantastic stuff with the defaults with this, so we only make a few minor tweaks.

```
(setq +org-msg-accent-color "#1a5fb4"
      org-msg-greeting-fmt "\nHi %s,\n\n"
      org-msg-signature "\n\n#+begin_signature\nAll the
      ↪  best,\\\\\n*Timothy*\n#+end_signature")
(map! :map org-msg-edit-mode-map
      :after org-msg
      :n "G" #'org-msg-goto-body)
```

6 Language configuration

6.1 General

6.1.1 File Templates

For some file types, we overwrite defaults in the snippets directory, others need to have a template assigned.

```
(set-file-template! "\\.tex$" :trigger "__" :mode 'latex-mode)
(set-file-template! "\\.org$" :trigger "__" :mode 'org-mode)
(set-file-template! "/LICEN[CS]E$" :trigger '+file-templates/insert-license)
```

6.2 Plaintext

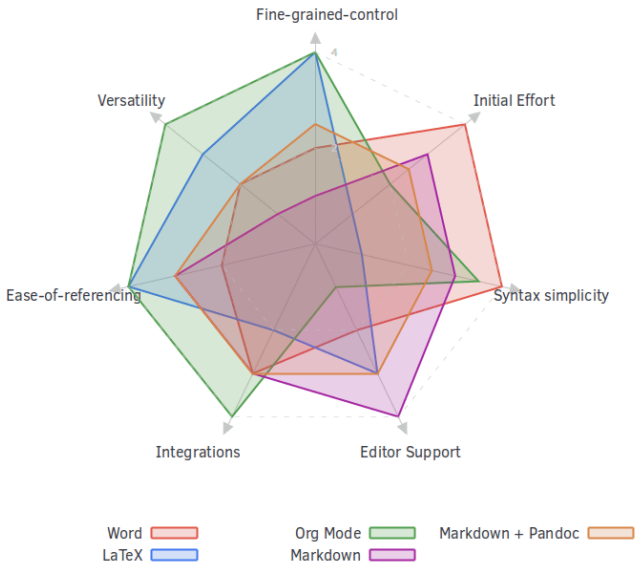
It's nice to see ANSI colour codes displayed

```
(after! text-mode
  (add-hook! 'text-mode-hook
             ;; Apply ANSI color codes
             (with-silent-modifications
               (ansi-color-apply-on-region (point-min) (point-max)))))
```

6.3 Org Mode

I really like org mode, I've given some thought to why, and below is the result.

Format	Fine-grained-control	Initial Effort	Syntax simplicity	Editor Support	Integ
Word	2	4	4	2	
LaTeX	4	1	1	3	
Org Mode	4	2	3.5	1	
Markdown	1	3	3	4	
Markdown + Pandoc	2.5	2.5	2.5	3	



Beyond the elegance in the markup language, tremendously rich integrations with Emacs allow for some fantastic features, such as what seems to be the best support for literate programming of any currently available technology.

```
      ¿¿¿Code¿¿¿            ¿¿¿Raw Code¿¿¿ Computer
Ideas¿¿        ¿¿¿ Org Mode¿¿
      ¿¿¿Text¿¿¿            ¿¿¿Document¿¿¿ People
```

An `.org` file can contain blocks of code (with noweb templating support), which can be tangled to dedicated source code files, and woven into a document (report, documentation, presentation, etc.) through various (*extensible*) methods. These source blocks may even create images or other content to be included in the document, or generate source code.

```
                  ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .pdf ¿
                 pdfLaTeX ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿                ¿
                  ¿     ¿                  ¿                ¿
                  ¿     ¿                  ¿                ¿
                 .tex    ¿                  ¿                ¿
                  ¿     ¿                  ¿                ¿
                ¿¿¿¿¿¿¿  ¿                  ¿ style.scss     ¿ Weaving
graphc.png ¿¿   ¿  embedded TeX             ¿      ¿         ¿ (Documents)
image.jpeg ¿¿ filters   ¿                   ¿    .css        ¿
            ¿     ¿     ¿                   ¿     ¿¿         ¿
figure.png¿¿¿¿¿ PROJECT.ORG ¿¿¿¿¿filters¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .html ¿
     ¿           ¿¿ ¿ ¿ ¿¿¿¿¿¿¿ embedded html ¿¿¿¿¿¿          ¿
     ¿¿¿¿¿¿¿¿¿¿¿¿¿ ¿ ¿                                     ¿
    result¿¿¿¿¿¿¿ ¿ ¿ ¿¿¿¿¿¿¿¿filters¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .txt  ¿
     ¿¿         ¿ ¿ ¿ ¿                                    ¿
    execution   ¿ ¿ ¿ ¿¿¿¿¿¿¿¿filters¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .md   ¿
     ¿¿         ¿ ¿ ¿
    code blocks¿¿ ¿ ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .c    ¿
     ¿¿¿¿¿¿¿¿¿¿¿¿¿ ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .sh   ¿ Tangling
                    ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .hs   ¿ (Code)
                    ¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿ .el   ¿
```

Finally, because this section is fairly expensive to initialise, we'll wrap it in an `(after! ...)` block.

```
(after! org
  <<org-conf>>
)
```

6.3.1 System config

Mime types Org mode isn't recognised as it's own mime type by default, but that can easily be changed with the following file. For system-wide changes try `/usr/share/mime/packages/org.xml`.

```
<mime-info xmlns='http://www.freedesktop.org/standards/shared-mime-info'>
  <mime-type type="text/org">
```

```
    <comment>Emacs Org-mode File</comment>
    <glob pattern="*.org"/>
    <alias type="text/org"/>
  </mime-type>
</mime-info>
```

What's nice is that Papirus now has an icon for `text/org`. One simply needs to refresh their mime database

```
update-mime-database ~/.local/share/mime
```

Then set Emacs as the default editor

```
xdg-mime default emacs.desktop text/org
```

Development Testing patches from the ML is currently more hassle than it needs to be. Let's change that.

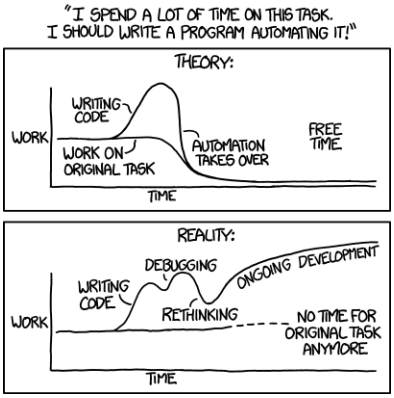
```
(defvar org-ml-target-dir "~/.emacs.d/.local/straight/repos/org-mode/")
(defvar org-ml-max-age 600
  "Maximum permissible age in seconds.")
(defvar org-ml--cache-timestamp 0)
(defvar org-ml--cache nil)

(defun org-ml-current-patches ()
  "Get the currently open patches, as a list of alists.
     Entries of the form (subject . id)."
  (delq nil
        (mapcar
         (lambda (entry)
           (unless (plist-get entry :fixed)
             (cons
              (format "%-8s  %s"
                      (propertize
                       (replace-regexp-in-string "T.*" ""
                                                 (plist-get entry :date))
                       'face 'font-lock-doc-face)
                      (propertize
                       (replace-regexp-in-string "\\[PATCH\\] ?" ""
                                                 (plist-get entry :summary))
                       'face 'font-lock-keyword-face))
              (plist-get entry :id))))
         (with-current-buffer (url-retrieve-synchronously
         ↪  "https://updates.orgmode.org/data/patches")
           (json-parse-buffer :object-type 'plist)))))

(defun org-ml-select-patch-thread ()
```

```
  "Find and apply a proposed Org patch."
  (interactive)
  (let ((current-workspace (+workspace-current))
        (patches (progn
                   (when (or (not org-ml--cache)
                             (> (- (float-time) org-ml--cache-timestamp)
                                org-ml-max-age))
                     (setq org-ml--cache (org-ml-current-patches)
                           org-ml--cache-timestamp (float-time)))
                   org-ml--cache))
        msg-id)
    (ivy-read "Thread: "
              patches
              :action (lambda (m) (setq msg-id (cdr m))))
    (+workspace-switch +mu4e-workspace-name)
    (mu4e-view-message-with-message-id msg-id)
    (add-to-list 'mu4e-view-actions
                 (cons "apply patch to org" #'org-ml-transient-mu4e-action))))

(defun org-ml-transient-mu4e-action (msg)
  (setq mu4e-view-actions
        (delete (cons "apply patch to org" #'org-ml-transient-mu4e-action)
                mu4e-view-actions))
  (+workspace/other)
  (magit-status org-ml-target-dir)
  (with-current-buffer (get-buffer-create "*Shell: Org apply patches*")
    (erase-buffer)
    (let ((default-directory org-ml-target-dir))
      (shell-command
       (format "git am %s"
               (shell-quote-argument (mu4e-message-field msg :path)))
       (current-buffer))
      (magit-refresh))
    (when (string-match-p "Error\\|failed" (buffer-string))
      (+popup/buffer))))
```

6.3.2 Behaviour

Tweaking defaults

```
(setq org-directory "~/.org"                    ; let's put files here
      org-use-property-inheritance t            ; it's convenient to have
      ↪ properties inherited
      org-log-done 'time                        ; having the time a item is done
      ↪ sounds convininet
      org-list-allow-alphabetical t             ; have a. A. a) A) list bullets
      org-export-in-background t                ; run export processes in external
      ↪ emacs process
      org-catch-invisible-edits 'smart          ; try not to accidently do weird
      ↪ stuff in invisible regions
```



Automation 'Automating' comes from the roots 'auto-' meaning 'self-', and 'mating', meaning 'screwing'.

```
      org-re-reveal-root "https://cdn.jsdelivr.net/npm/reveal.js")
```

I also like the `:comments` header-argument, so let's make that a default.

```
(setq org-babel-default-header-args
      '((:session . "none")
        (:results . "replace")
        (:exports . "code")
        (:cache . "no")
        (:noweb . "no")
        (:hlines . "no")
        (:tangle . "no")
        (:comments . "link")))
```

By default, `visual-line-mode` is turned on, and `auto-fill-mode` `off` by a hook. However this messes with tables in Org-mode, and other plaintext files (e.g. markdown, LaTeX) so I'll turn it off for this, and manually enable it for more specific modes as desired.

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

There also seem to be a few keybindings which use `hjkl`, but miss arrow key equivalents.

```
(map! :map evil-org-mode-map
      :after evil-org
      :n "g <up>" #'org-backward-heading-same-level
      :n "g <down>" #'org-forward-heading-same-level
      :n "g <left>" #'org-up-element
      :n "g <right>" #'org-down-element)
```

Extra functionality

Org buffer creation Let's also make creating an org buffer just that little bit easier.

```
(evil-define-command evil-buffer-org-new (count file)
  "Creates a new ORG buffer replacing the current window, optionally
         editing a certain FILE"
  :repeat nil
  (interactive "P<f>")
  (if file
      (evil-edit file)
    (let ((buffer (generate-new-buffer "*new org*")))
      (set-window-buffer nil buffer)
      (with-current-buffer buffer
        (org-mode)))))
(map! :leader
      (:prefix "b"
       :desc "New empty ORG buffer" "o" #'evil-buffer-org-new))
```

List bullet sequence I think it makes sense to have list bullets change with depth

```
(setq org-list-demote-modify-bullet '(("+" . "-") ("-" . "+") ("*" . "+") ("1." .
↪  "a.")))
```

Citation Occasionally I want to cite something.

```
(use-package! org-ref
  :after org
  :config
  (setq org-ref-completion-library 'org-ref-ivy-cite))
```

cdlatex It's also nice to be able to use `cdlatex`.

```
(add-hook 'org-mode-hook 'turn-on-org-cdlatex)
```

It's handy to be able to quickly insert environments with `C-c }`. I almost always want to edit them afterwards though, so let's make that happen by default.

```
(defadvice! org-edit-latex-emv-after-insert ()
  :after #'org-cdlatex-environment-indent
  (org-edit-latex-environment))
```

At some point in the future it could be good to investigate splitting org blocks. Likewise this looks good for symbols.

Spellcheck My spelling is atrocious, so let's get flycheck going.

```
(add-hook 'org-mode-hook 'turn-on-flyspell)
```

LSP support in `src` blocks Now, by default, LSPs don't really function at all in `src` blocks.

```
(cl-defmacro lsp-org-babel-enable (lang)
  "Support LANG in org source code block."
  (setq centaur-lsp 'lsp-mode)
  (cl-check-type lang stringp)
  (let* ((edit-pre (intern (format "org-babel-edit-prep:%s" lang)))
         (intern-pre (intern (format "lsp--%s" (symbol-name edit-pre)))))
    `(progn
       (defun ,intern-pre (info)
         (let ((file-name (->> info caddr (alist-get :file))))
           (unless file-name
             (setq file-name (make-temp-file "babel-lsp-")))
           (setq buffer-file-name file-name)
           (lsp-deferred)))
       (put ',intern-pre 'function-documentation
            (format "Enable lsp-mode in the buffer of org source block (%s)."
                    (upcase ,lang)))
       (if (fboundp ',edit-pre)
           (advice-add ',edit-pre :after ',intern-pre)
         (progn
           (defun ,edit-pre (info)
             (,intern-pre info))
           (put ',edit-pre 'function-documentation
                (format "Prepare local buffer environment for org source block
                ↪  (%s)."
                        (upcase ,lang))))))))
(defvar org-babel-lang-list
  '("go" "python" "ipython" "bash" "sh"))
(dolist (lang org-babel-lang-list)
  (eval `(lsp-org-babel-enable ,lang)))
```

View exported file `'localeader v` has no pre-existing binding, so I may as well use it with the same functionality as in LaTeX. Let's try viewing possible output files with this.

```
(map! :map org-mode-map
      :localleader
      :desc "View exported file" "v" #'org-view-output-file)

(defun org-view-output-file (&optional org-file-path)
  "Visit buffer open on the first output file (if any) found, using
  ↪ `org-view-output-file-extensions'"
  (interactive)
  (let* ((org-file-path (or org-file-path (buffer-file-name) ""))
         (dir (file-name-directory org-file-path))
         (basename (file-name-base org-file-path))
         (output-file nil))
    (dolist (ext org-view-output-file-extensions)
      (unless output-file
        (when (file-exists-p
               (concat dir basename "." ext))
          (setq output-file (concat dir basename "." ext)))))
    (if output-file
        (if (member (file-name-extension output-file)
        ↪ org-view-external-file-extensions)
            (browse-url-xdg-open output-file)
          (pop-to-buffer (or (find-buffer-visiting output-file)
                             (find-file-noselect output-file))))
      (message "No exported file found"))))

(defvar org-view-output-file-extensions '("pdf" "md" "rst" "txt" "tex" "html")
  "Search for output files with these extensions, in order, viewing the first that
  ↪ matches")
(defvar org-view-external-file-extensions '("html")
  "File formats that should be opened externally.")
```

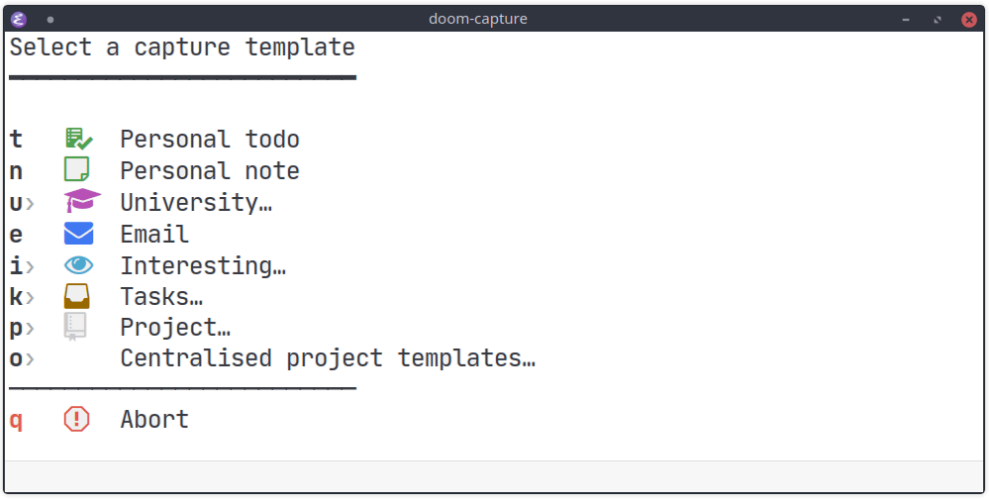
Super agenda

```
(use-package! org-super-agenda
  :commands (org-super-agenda-mode))
(after! org-agenda
  (org-super-agenda-mode))

(setq org-agenda-skip-scheduled-if-done t
      org-agenda-skip-deadline-if-done t
      org-agenda-include-deadlines t
      org-agenda-block-separator nil
      org-agenda-tags-column 100 ;; from testing this seems to be a good value
      org-agenda-compact-blocks t)

(setq org-agenda-custom-commands
      '(("o" "Overview"
         ((agenda "" ((org-agenda-span 'day)
```

```
                    (org-super-agenda-groups
                     '((:name "Today"
                        :time-grid t
                        :date today
                        :todo "TODAY"
                        :scheduled today
                        :order 1)))))
          (alltodo "" ((org-agenda-overriding-header "")
                       (org-super-agenda-groups
                        '((:name "Next to do"
                           :todo "NEXT"
                           :order 1)
                          (:name "Important"
                           :tag "Important"
                           :priority "A"
                           :order 6)
                          (:name "Due Today"
                           :deadline today
                           :order 2)
                          (:name "Due Soon"
                           :deadline future
                           :order 8)
                          (:name "Overdue"
                           :deadline past
                           :face error
                           :order 7)
                          (:name "Assignments"
                           :tag "Assignment"
                           :order 10)
                          (:name "Issues"
                           :tag "Issue"
                           :order 12)
                          (:name "Emacs"
                           :tag "Emacs"
                           :order 13)
                          (:name "Projects"
                           :tag "Project"
                           :order 14)
                          (:name "Research"
                           :tag "Research"
                           :order 15)
                          (:name "To read"
                           :tag "Read"
                           :order 30)
                          (:name "Waiting"
                           :todo "WAITING"
                           :order 20)
                          (:name "University"
                           :tag "uni"
                           :order 32)
                          (:name "Trivial"
                           :priority<= "E"
                           :tag ("Trivial" "Unimportant")
```

```
                            :todo ("SOMEDAY" )
                            :order 90)
                           (:discard (:tag ("Chore" "Routine" "Daily")))))))))))
```

Capture Let's setup some org-capture templates, and make them visually nice to access.




```
(use-package! doct
  :commands (doct))

(after! org-capture
  <<prettify-capture>>
  (setq +org-capture-uni-units (condition-case nil
                                   (split-string (f-read-text "~/.org/.uni-units"))
                                 (error nil))
        +org-capture-recipies  "~/Desktop/TEC/Organisation/recipies.org")

  (defun +doct-icon-declaration-to-icon (declaration)
    "Convert :icon declaration to icon"
    (let ((name (pop declaration))
          (set  (intern (concat "all-the-icons-" (plist-get declaration :set))))
          (face (intern (concat "all-the-icons-" (plist-get declaration :color))))
          (v-adjust (or (plist-get declaration :v-adjust) 0.01)))
      (apply set `(,name :face ,face :v-adjust ,v-adjust))))

  (defun +doct-iconify-capture-templates (groups)
    "Add declaration's :icon to each template group in GROUPS."
    (let ((templates (doct-flatten-lists-in groups)))
      (setq doct-templates (mapcar (lambda (template)
                                     (when-let* ((props (nthcdr (if (= (length
                                     ↪  template) 4) 2 5) template))
```

```
                                                (spec (plist-get (plist-get props
                                                ↪  :doct) :icon)))
                                          (setf (nth 1 template) (concat
                                          ↪  (+doct-icon-declaration-to-icon spec)
                                                                         "\t"
                                                                         (nth 1
                                                                         ↪  template))))
                                        template)
                                      templates))))

(setq doct-after-conversion-functions '(+doct-iconify-capture-templates))

(defun set-org-capture-templates ()
  (setq org-capture-templates
        (doct `(("Personal todo" :keys "t"
                 :icon ("checklist" :set "octicon" :color "green")
                 :file +org-capture-todo-file
                 :prepend t
                 :headline "Inbox"
                 :type entry
                 :template ("* TODO %?"
                            "%i %a")
                 )
                ("Personal note" :keys "n"
                 :icon ("sticky-note-o" :set "faicon" :color "green")
                 :file +org-capture-todo-file
                 :prepend t
                 :headline "Inbox"
                 :type entry
                 :template ("* %?"
                            "%i %a")
                 )
                ("University" :keys "u"
                 :icon ("graduation-cap" :set "faicon" :color "purple")
                 :file +org-capture-todo-file
                 :headline "University"
                 :unit-prompt ,(format "%%^{Unit|%s}" (string-join
                 ↪  +org-capture-uni-units "|"))
                 :prepend t
                 :type entry
                 :children (("Test" :keys "t"
                             :icon ("timer" :set "material" :color "red")
                             :template ("* TODO [#C] %{unit-prompt} %?
                             ↪  :uni:tests:"
                                        "SCHEDULED: %^{Test date:}T"
                                        "%i %a"))
                            ("Assignment" :keys "a"
                             :icon ("library_books" :set "material" :color
                             ↪  "orange")
                             :template ("* TODO [#B] %{unit-prompt} %?
                             ↪  :uni:assignments:"
                                        "DEADLINE: %^{Due date:}T"
                                        "%i %a"))
```

```
                ("Lecture" :keys "l"
                 :icon ("keynote" :set "fileicon" :color "orange")
                 :template ("* TODO [#C] %{unit-prompt} %?
                 ↪  :uni:lecture:"
                           "%i %a"))
                ("Miscellaneous task" :keys "u"
                 :icon ("list" :set "faicon" :color "yellow")
                 :template ("* TODO [#D] %{unit-prompt} %? :uni:"
                           "%i %a"))))
        ("Email" :keys "e"
         :icon ("envelope" :set "faicon" :color "blue")
         :file +org-capture-todo-file
         :prepend t
         :headline "Inbox"
         :type entry
         :template ("* TODO %^{type|reply to|contact} %\\3 %? :email:"
                    "Send an email %^{urgancy|soon|ASAP|anon|at some
                    ↪  point|eventually} to %^{recipiant}"
                    "about %^{topic}"
                    "%U %i %a"))
        ("Interesting" :keys "i"
         :icon ("eye" :set "faicon" :color "lcyan")
         :file +org-capture-todo-file
         :prepend t
         :headline "Interesting"
         :type entry
         :template ("* [ ] %{desc}%? :%{i-type}:"
                    "%i %a")
         :children (("Webpage" :keys "w"
                     :icon ("globe" :set "faicon" :color "green")
                     :desc "%(org-cliplink-capture) "
                     :i-type "read:web"
                     )
                    ("Article" :keys "a"
                     :icon ("file-text" :set "octicon" :color "yellow")
                     :desc ""
                     :i-type "read:reaserch"
                     )
                    ("\tRecipie" :keys "r"
                     :icon ("spoon" :set "faicon" :color "dorange")
                     :file +org-capture-recipies
                     :headline "Unsorted"
                     :template "%(org-chef-get-recipe-from-url)"
                     )
                    ("Information" :keys "i"
                     :icon ("info-circle" :set "faicon" :color "blue")
                     :desc ""
                     :i-type "read:info"
                     )
                    ("Idea" :keys "I"
                     :icon ("bubble_chart" :set "material" :color
                     ↪  "silver")
                     :desc ""
```

```
                :i-type "idea"
                )))
            ("Tasks" :keys "k"
             :icon ("inbox" :set "octicon" :color "yellow")
             :file +org-capture-todo-file
             :prepend t
             :headline "Tasks"
             :type entry
             :template ("* TODO %? %^G%{extra}"
                        "%i %a")
             :children (("General Task" :keys "k"
                         :icon ("inbox" :set "octicon" :color "yellow")
                         :extra ""
                         )
                        ("Task with deadline" :keys "d"
                         :icon ("timer" :set "material" :color "orange"
                         ↪  :v-adjust -0.1)
                         :extra "\nDEADLINE: %^{Deadline:}t"
                         )
                        ("Scheduled Task" :keys "s"
                         :icon ("calendar" :set "octicon" :color "orange")
                         :extra "\nSCHEDULED: %^{Start time:}t"
                         )
                        ))
            ("Project" :keys "p"
             :icon ("repo" :set "octicon" :color "silver")
             :prepend t
             :type entry
             :headline "Inbox"
             :template ("* %{time-or-todo} %?"
                        "%i"
                        "%a")
             :file ""
             :custom (:time-or-todo "")
             :children (("Project-local todo" :keys "t"
                         :icon ("checklist" :set "octicon" :color "green")
                         :time-or-todo "TODO"
                         :file +org-capture-project-todo-file)
                        ("Project-local note" :keys "n"
                         :icon ("sticky-note" :set "faicon" :color "yellow")
                         :time-or-todo "%U"
                         :file +org-capture-project-notes-file)
                        ("Project-local changelog" :keys "c"
                         :icon ("list" :set "faicon" :color "blue")
                         :time-or-todo "%U"
                         :heading "Unreleased"
                         :file +org-capture-project-changelog-file))
             )
            ("\tCentralised project templates"
             :keys "o"
             :type entry
             :prepend t
             :template ("* %{time-or-todo} %?"
```

```
                                "%i"
                                "%a")
                    :children (("Project todo"
                                :keys "t"
                                :prepend nil
                                :time-or-todo "TODO"
                                :heading "Tasks"
                                :file +org-capture-central-project-todo-file)
                               ("Project note"
                                :keys "n"
                                :time-or-todo "%U"
                                :heading "Notes"
                                :file +org-capture-central-project-notes-file)
                               ("Project changelog"
                                :keys "c"
                                :time-or-todo "%U"
                                :heading "Unreleased"
                                :file +org-capture-central-project-changelog-file))
                    )))))

  (set-org-capture-templates)
  (unless (display-graphic-p)
    (add-hook 'server-after-make-frame-hook
              (defun org-capture-reinitialise-hook ()
                (when (display-graphic-p)
                  (set-org-capture-templates)
                  (remove-hook 'server-after-make-frame-hook
                               #'org-capture-reinitialise-hook))))))
```

It would also be nice to improve how the capture dialogue looks

```
(defun org-capture-select-template-prettier (&optional keys)
  "Select a capture template, in a prettier way than default
      Lisp programs can force the template by setting KEYS to a string."
  (let ((org-capture-templates
         (or (org-contextualize-keys
              (org-capture-upgrade-templates org-capture-templates)
              org-capture-templates-contexts)
             '(("t" "Task" entry (file+headline "" "Tasks")
                "* TODO %?\n  %u\n  %a")))))
    (if keys
        (or (assoc keys org-capture-templates)
            (error "No capture template referred to by \"%s\" keys" keys))
      (org-mks org-capture-templates
               "Select a capture template\n¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿"
               "Template key: "
               `(("q" ,(concat (all-the-icons-octicon "stop" :face
               ↪  'all-the-icons-red :v-adjust 0.01) "\tAbort")))))))
(advice-add 'org-capture-select-template :override
↪  #'org-capture-select-template-prettier)
```

```
(defun org-mks-pretty (table title &optional prompt specials)
  "Select a member of an alist with multiple keys. Prettified.
     TABLE is the alist which should contain entries where the car is a string.
     There should be two types of entries.
     1. prefix descriptions like (\"a\" \"Description\")
        This indicates that `a' is a prefix key for multi-letter selection, and
        that there are entries following with keys like \"ab\", \"ax\"¿
     2. Select-able members must have more than two elements, with the first
        being the string of keys that lead to selecting it, and the second a
        short description string of the item.
     The command will then make a temporary buffer listing all entries
     that can be selected with a single key, and all the single key
     prefixes.  When you press the key for a single-letter entry, it is selected.
     When you press a prefix key, the commands (and maybe further prefixes)
     under this key will be shown and offered for selection.
     TITLE will be placed over the selection in the temporary buffer,
     PROMPT will be used when prompting for a key.  SPECIALS is an
     alist with (\"key\" \"description\") entries.  When one of these
     is selected, only the bare key is returned."
  (save-window-excursion
    (let ((inhibit-quit t)
          (buffer (org-switch-to-buffer-other-window "*Org Select*"))
          (prompt (or prompt "Select: "))
          case-fold-search
          current)
      (unwind-protect
          (catch 'exit
            (while t
              (setq-local evil-normal-state-cursor (list nil))
              (erase-buffer)
              (insert title "\n\n")
              (let ((des-keys nil)
                    (allowed-keys '("\C-g"))
                    (tab-alternatives '("\s" "\t" "\r"))
                    (cursor-type nil))
                ;; Populate allowed keys and descriptions keys
                ;; available with CURRENT selector.
                (let ((re (format "\\`%s\\(.\\)\\'"
                                  (if current (regexp-quote current) "")))
                      (prefix (if current (concat current " ") "")))
                  (dolist (entry table)
                    (pcase entry
                      ;; Description.
                      (`(,(and key (pred (string-match re))) ,desc)
                       (let ((k (match-string 1 key)))
                         (push k des-keys)
                         ;; Keys ending in tab, space or RET are equivalent.
                         (if (member k tab-alternatives)
                             (push "\t" allowed-keys)
                           (push k allowed-keys))
                         (insert (propertize prefix 'face 'font-lock-comment-face)
                         ↪  (propertize k 'face 'bold) (propertize "¿" 'face
                         ↪  'font-lock-comment-face) "  " desc "¿" "\n")))
```

```
                    ;; Usable entry.
                    (`(,(and key (pred (string-match re))) ,desc . ,_)
                     (let ((k (match-string 1 key)))
                       (insert (propertize prefix 'face 'font-lock-comment-face)
                       ↪  (propertize k 'face 'bold) "   " desc "\n")
                       (push k allowed-keys)))
                    (_ nil))))
                ;; Insert special entries, if any.
                (when specials
                  (insert "¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿¿\n")
                  (pcase-dolist (`(,key ,description) specials)
                    (insert (format "%s   %s\n" (propertize key 'face '(bold
                    ↪  all-the-icons-red)) description))
                    (push key allowed-keys)))
                ;; Display UI and let user select an entry or
                ;; a sub-level prefix.
                (goto-char (point-min))
                (unless (pos-visible-in-window-p (point-max))
                  (org-fit-window-to-buffer))
                (let ((pressed (org--mks-read-key allowed-keys prompt)))
                  (setq current (concat current pressed))
                  (cond
                   ((equal pressed "\C-g") (user-error "Abort"))
                   ;; Selection is a prefix: open a new menu.
                   ((member pressed des-keys))
                   ;; Selection matches an association: return it.
                   ((let ((entry (assoc current table)))
                      (and entry (throw 'exit entry))))
                   ;; Selection matches a special entry: return the
                   ;; selection prefix.
                   ((assoc current specials) (throw 'exit current))
                   (t (error "No entry available")))))))
        (when buffer (kill-buffer buffer))))))
(advice-add 'org-mks :override #'org-mks-pretty)
```

The org-capture bin is rather nice, but I'd be nicer with a smaller frame, and no modeline.

```
(setf (alist-get 'height +org-capture-frame-parameters) 15)
;; (alist-get 'name +org-capture-frame-parameters) "¿ Capture") ;; ATM hardcoded in
↪  other places, so changing breaks stuff
(setq +org-capture-fn
      (lambda ()
        (interactive)
        (set-window-parameter nil 'mode-line-format 'none)
        (org-capture)))
```

Roam

Basic settings I'll just set this to be within `Organisation` folder for now, in the future it could be worth seeing if I could hook this up to a Nextcloud instance.

```
(setq org-roam-directory "~/Desktop/TEC/Organisation/Roam/")
```

That said, if the directory doesn't exist we likely don't want to be using roam. Since we don't want to trigger errors (which will happen as soon as roam tries to initialise), let's not load roam.

```
(package! org-roam :disable t)
```

Registering roam protocol The recommended method of registering a protocol is by registering a desktop application, which seems reasonable.

```
[Desktop Entry]
Name=Org-Protocol
Exec=emacsclient %u
Icon=emacs-icon
Type=Application
Terminal=false
MimeType=x-scheme-handler/org-protocol
```

To associate `org-protocol://` links with the desktop file,

```
xdg-mime default org-protocol.desktop x-scheme-handler/org-protocol
```

Graph Behaviour By default, clicking on an `org-protocol://` link messes with the `svg` view. To fix this we can use an `iframe`, however that requires shifting to an `html` file. Hence, we need to do a bit of overriding.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Roam Graph</title>
    <meta name="viewport" content="width=device-width">
    <style type="text/css">
      body {
      background: white;
      }

      svg {
```

```
      position: relative;
      top: 50vh;
      left: 50vw;
      transform: translate(-50%, -50%);
      width: 95vw;
      }

      a > polygon {
      transition-duration: 200ms;
      transition-property: fill;
      }

      a > polyline {
      transition-duration: 400ms;
      transition-property: stroke;
      }

      a:hover > polygon {
      fill: #d4d4d4;
      }
      a:hover > polyline {
      stroke: #888;
      }
    </style>
    <script>
      function create_iframe (url) {
      i = document.createElement('iframe');
      i.setAttribute('src', url);
      i.style.setProperty('display', 'none');
      document.body.append(i);
      }
      function listen_on_all_a () {
      document.querySelectorAll("svg a").forEach(elem => {
      elem.addEventListener('click', (e) => {
      e.preventDefault();
      create_iframe(elem.href.baseVal);
      });
      });
      }
    </script>
  </head>
  <body onload="listen_on_all_a()">
    %s
  </body>
</html>
```

```
(after! org-roam
  (setq org-roam-graph-node-extra-config
        '(("shape"      . "underline")
          ("style"      . "rounded,filled")
          ("fillcolor"  . "#EEEEEE")
          ("color"      . "#C9C9C9")
```

```
          ("fontcolor"   . "#111111")
          ("fontname"    . "Overpass")))

  (setq +org-roam-graph--html-template
        (replace-regexp-in-string "%\\([^s]\\)" "%%\\1"
                                  (f-read-text (concat doom-private-dir
                                  ↪  "misc/org-roam-template.html"))))

  (defadvice! +org-roam-graph--build-html (&optional node-query callback)
    "Generate a graph showing the relations between nodes in NODE-QUERY. HTML
    ↪  style."
    :override #'org-roam-graph--build
    (unless (stringp org-roam-graph-executable)
      (user-error "`org-roam-graph-executable' is not a string"))
    (unless (executable-find org-roam-graph-executable)
      (user-error (concat "Cannot find executable %s to generate the graph.  "
                          "Please adjust `org-roam-graph-executable'")
                  org-roam-graph-executable))
    (let* ((node-query (or node-query
                           `[:select [file titles] :from titles
                             ,@(org-roam-graph--expand-matcher 'file t)]))
           (graph      (org-roam-graph--dot node-query))
           (temp-dot   (make-temp-file "graph." nil ".dot" graph))
           (temp-graph (make-temp-file "graph." nil ".svg"))
           (temp-html  (make-temp-file "graph." nil ".html")))
      (org-roam-message "building graph")
      (make-process
       :name "*org-roam-graph--build-process*"
       :buffer "*org-roam-graph--build-process*"
       :command `(,org-roam-graph-executable ,temp-dot "-Tsvg" "-o" ,temp-graph)
       :sentinel (progn
                   (lambda (process _event)
                     (when (= 0 (process-exit-status process))
                       (write-region (format +org-roam-graph--html-template
                       ↪  (f-read-text temp-graph)) nil temp-html)
                       (when callback
                         (funcall callback temp-html)))))))))
```

Modeline file name All those numbers! It's messy. Let's adjust this in a similar way that I have in the Window title.

```
(defadvice! doom-modeline--buffer-file-name-roam-aware-a (orig-fun)
  :around #'doom-modeline-buffer-file-name ; takes no args
  (if (s-contains-p org-roam-directory (or buffer-file-name ""))
      (replace-regexp-in-string

       ↪  "\\(?:^\\|.*/\\)\\([0-9]\\{4\\}\\)\\([0-9]\\{2\\}\\)\\([0-9]\\{2\\}\\)[0-9]*-"
       "¿(\\1-\\2-\\3) "
       (subst-char-in-string ?_ ?  buffer-file-name))
    (funcall orig-fun)))
```

Nicer generated heading IDs Thanks to alphapapa's unpackaged.el.

By default, `url-hexify-string` seemed to cause me some issues. Replacing that in `a53899` resolved this for me. To go one step further, I create a function for producing nice short links, like an inferior version of `reftex-label`.

```
(defvar org-reference-contraction-max-words 3
  "Maximum number of words in a reference reference.")
(defvar org-reference-contraction-max-length 35
  "Maximum length of resulting reference reference, including joining characters.")
(defvar org-reference-contraction-stripped-words
  '("the" "on" "in" "off" "a" "for" "by" "of" "and" "is" "to")
  "Superfluous words to be removed from a reference.")
(defvar org-reference-contraction-joining-char "-"
  "Character used to join words in the reference reference.")

(defun org-reference-contraction-truncate-words (words)
  "Using `org-reference-contraction-max-length' as the total character 'budget' for
      the WORDS
      and truncate individual words to conform to this budget.
      To arrive at a budget that accounts for words undershooting their requisite
      average length,
      the number of charachters in the budget freed by short words is distributed
      among the words
      exceeding the average length.  This adjusts the per-word budget to be the
      maximum feasable for
      this particular situation, rather than the universal maximum average.
      This budget-adjusted per-word maximum length is given by the mathematical
      expression below:
      max length = \\floor{ \\frac{total length - chars for seperators - \\sum_{word
  ↪  \\leq average length} length(word) }{num(words) > average length} }"
  ;; trucate each word to a max word length determined by
  ;;
  (let* ((total-length-budget (- org-reference-contraction-max-length  ; how many
  ↪  non-separator chars we can use
                                 (1- (length words))))
         (word-length-budget (/ total-length-budget                     ; max
         ↪  length of each word to keep within budget
                                org-reference-contraction-max-words))
         (num-overlong (-count (lambda (word)                           ; how many
         ↪  words exceed that budget
                                 (> (length word) word-length-budget))
                               words))
         (total-short-length (-sum (mapcar (lambda (word)               ; total
         ↪  length of words under that budget
                                             (if (<= (length word)
                                             ↪  word-length-budget)
                                                 (length word) 0))
                                           words)))
         (max-length (/ (- total-length-budget total-short-length)      ;
         ↪  max(max-length) that we can have to fit within the budget
                        num-overlong)))
```

```
    (mapcar (lambda (word)
              (if (<= (length word) max-length)
                  word
                (substring word 0 max-length)))
            words)))

(defun org-reference-contraction (reference-string)
  "Give a contracted form of REFERENCE-STRING that is only contains alphanumeric
      characters.
      Strips 'joining' words present in `org-reference-contraction-stripped-words',
      and then limits the result to the first `org-reference-contraction-max-words'
      words.
      If the total length is > `org-reference-contraction-max-length' then
      individual words are
      truncated to fit within the limit using
  ↪   `org-reference-contraction-truncate-words'."
  (let ((reference-words
         (-filter (lambda (word)
                    (not (member word org-reference-contraction-stripped-words)))
                  (split-string
                   (->> reference-string
                        downcase
                        (replace-regexp-in-string
                        ↪  "\\[\\[[^]]+\\]\\[\\([^]]+\\)\\]\\]" "\\1") ; get
                        ↪  description from org-link
                        (replace-regexp-in-string "[-/ ]+" " ") ; replace
                        ↪  seperator-type chars with space
                        (replace-regexp-in-string "[^a-z0-9 ]" "") ; strip chars
                        ↪  which need %-encoding in a uri
                        ) " "))))
    (when (> (length reference-words)
             org-reference-contraction-max-words)
      (setq reference-words
            (cl-subseq reference-words 0 org-reference-contraction-max-words)))

    (when (> (apply #'+ (1- (length reference-words))
                    (mapcar #'length reference-words))
             org-reference-contraction-max-length)
      (setq reference-words (org-reference-contraction-truncate-words
      ↪  reference-words)))

    (string-join reference-words org-reference-contraction-joining-char)))
```

Now here's alphapapa's subtly tweaked mode.

```
(define-minor-mode unpackaged/org-export-html-with-useful-ids-mode
  "Attempt to export Org as HTML with useful link IDs.
      Instead of random IDs like \"#orga1b2c3\", use heading titles,
      made unique when necessary."
  :global t
  (if unpackaged/org-export-html-with-useful-ids-mode
```

```
      (advice-add #'org-export-get-reference :override
      ↪  #'unpackaged/org-export-get-reference)
    (advice-remove #'org-export-get-reference
    ↪  #'unpackaged/org-export-get-reference)))
(unpackaged/org-export-html-with-useful-ids-mode 1) ; ensure enabled, and advice run

(defun unpackaged/org-export-get-reference (datum info)
  "Like `org-export-get-reference', except uses heading titles instead of random
  ↪  numbers."
  (let ((cache (plist-get info :internal-references)))
    (or (car (rassq datum cache))
        (let* ((crossrefs (plist-get info :crossrefs))
               (cells (org-export-search-cells datum))
               ;; Preserve any pre-existing association between
               ;; a search cell and a reference, i.e., when some
               ;; previously published document referenced a location
               ;; within current file (see
               ;; `org-publish-resolve-external-link').
               ;;
               ;; However, there is no guarantee that search cells are
               ;; unique, e.g., there might be duplicate custom ID or
               ;; two headings with the same title in the file.
               ;;
               ;; As a consequence, before re-using any reference to
               ;; an element or object, we check that it doesn't refer
               ;; to a previous element or object.
               (new (or (cl-some
                         (lambda (cell)
                           (let ((stored (cdr (assoc cell crossrefs))))
                             (when stored
                               (let ((old (org-export-format-reference stored)))
                                 (and (not (assoc old cache)) stored)))))
                         cells)
                        (when (org-element-property :raw-value datum)
                          ;; Heading with a title
                          (unpackaged/org-export-new-named-reference datum cache))
                        (when (member (car datum) '(src-block table example
                        ↪  fixed-width property-drawer))
                          ;; Nameable elements
                          (unpackaged/org-export-new-named-reference datum cache))
                        ;; NOTE: This probably breaks some Org Export
                        ;; feature, but if it does what I need, fine.
                        (org-export-format-reference
                         (org-export-new-reference cache))))
               (reference-string new))
          ;; Cache contains both data already associated to
          ;; a reference and in-use internal references, so as to make
          ;; unique references.
          (dolist (cell cells) (push (cons cell new) cache))
          ;; Retain a direct association between reference string and
          ;; DATUM since (1) not every object or element can be given
          ;; a search cell (2) it permits quick lookup.
          (push (cons reference-string datum) cache)
```

```
          (plist-put info :internal-references cache)
          reference-string))))

(defun unpackaged/org-export-new-named-reference (datum cache)
  "Return new reference for DATUM that is unique in CACHE."
  (cl-macrolet ((inc-suffixf (place)
                             `(progn
                                (string-match (rx bos
                                                  (minimal-match (group (1+
                                                  ↪  anything)))
                                                  (optional "--" (group (1+ digit)))
                                                  eos)
                                              ,place)
                                ;; HACK: `s1' instead of a gensym.
                                (-let* (((s1 suffix) (list (match-string 1 ,place)
                                                           (match-string 2 ,place)))
                                        (suffix (if suffix
                                                    (string-to-number suffix)
                                                  0)))
                                  (setf ,place (format "%s--%s" s1 (cl-incf
                                                  ↪  suffix)))))))
    (let* ((headline-p (eq (car datum) 'headline))
           (title (if headline-p
                      (org-element-property :raw-value datum)
                    (or (org-element-property :name datum)
                        (concat (org-element-property :raw-value
                                                      (org-element-property :parent

                                                                            ↪  (org-element-property
                                                                            ↪  :parent
                                                                            ↪  datum)))))))
           ;; get ascii-only form of title without needing percent-encoding
           (ref (concat (org-reference-contraction (substring-no-properties title))
                        (unless (or headline-p (org-element-property :name datum))
                          (concat ","
                                  (pcase (car datum)
                                    ('src-block "code")
                                    ('example "example")
                                    ('fixed-width "mono")
                                    ('property-drawer "properties")
                                    (_ (symbol-name (car datum))))
                                  "--1"))))
           (parent (when headline-p (org-element-property :parent datum))))
      (while (--any (equal ref (car it))
                    cache)
        ;; Title not unique: make it so.
        (if parent
            ;; Append ancestor title.
            (setf title (concat (org-element-property :raw-value parent)
                                "--" title)
                  ;; get ascii-only form of title without needing percent-encoding
                  ref (org-reference-contraction (substring-no-properties title))
                  parent (when headline-p (org-element-property :parent parent)))
```

```
          ;; No more ancestors: add and increment a number.
          (inc-suffixf ref)))
      ref)))

(add-hook 'org-load-hook #'unpackaged/org-export-html-with-useful-ids-mode)
```

Nicer `org-return` Once again, from unpackaged.el

```
(defun unpackaged/org-element-descendant-of (type element)
  "Return non-nil if ELEMENT is a descendant of TYPE.
      TYPE should be an element type, like `item' or `paragraph'.
      ELEMENT should be a list like that returned by `org-element-context'."
  ;; MAYBE: Use `org-element-lineage'.
  (when-let* ((parent (org-element-property :parent element)))
    (or (eq type (car parent))
        (unpackaged/org-element-descendant-of type parent))))

;;;###autoload
(defun unpackaged/org-return-dwim (&optional default)
  "A helpful replacement for `org-return-indent'.  With prefix, call
      `org-return-indent'.
      On headings, move point to position after entry content.  In
      lists, insert a new item or end the list, with checkbox if
      appropriate.  In tables, insert a new row or end the table."
  ;; Inspired by John Kitchin:
  ;; http://kitchingroup.cheme.cmu.edu/blog/2017/04/09/A-better-return-in-org-mode/
  (interactive "P")
  (if default
      (org-return t)
    (cond
     ;; Act depending on context around point.

     ;; NOTE: I prefer RET to not follow links, but by uncommenting this block,
     ;; links will be
     ;; followed.

     ;; ((eq 'link (car (org-element-context)))
     ;;  ;; Link: Open it.
     ;;  (org-open-at-point-global))

     ((org-at-heading-p)
      ;; Heading: Move to position after entry content.
      ;; NOTE: This is probably the most interesting feature of this function.
      (let ((heading-start (org-entry-beginning-position)))
        (goto-char (org-entry-end-position))
        (cond ((and (org-at-heading-p)
                    (= heading-start (org-entry-beginning-position)))
               ;; Entry ends on its heading; add newline after
               (end-of-line)
               (insert "\n\n"))
```

```
            (t
             ;; Entry ends after its heading; back up
             (forward-line -1)
             (end-of-line)
             (when (org-at-heading-p)
               ;; At the same heading
               (forward-line)
               (insert "\n")
               (forward-line -1))
             ;; FIXME: looking-back is supposed to be called with more arguments.
             (while (not (looking-back (rx (repeat 3 (seq (optional blank)
             ↪  "\n")))))
               (insert "\n"))
             (forward-line -1)))))

    ((org-at-item-checkbox-p)
     ;; Checkbox: Insert new item with checkbox.
     (org-insert-todo-heading nil))

    ((org-in-item-p)
     ;; Plain list.  Yes, this gets a little complicated...
     (let ((context (org-element-context)))
       (if (or (eq 'plain-list (car context))  ; First item in list
               (and (eq 'item (car context))
                    (not (eq (org-element-property :contents-begin context)
                             (org-element-property :contents-end context))))
               (unpackaged/org-element-descendant-of 'item context))  ; Element in
               ↪  list item, e.g. a link
           ;; Non-empty item: Add new item.
           (org-insert-item)
         ;; Empty item: Close the list.
         ;; TODO: Do this with org functions rather than operating on the text.
         ↪  Can't seem to find the right function.
         (delete-region (line-beginning-position) (line-end-position))
         (insert "\n"))))

    ((when (fboundp 'org-inlinetask-in-task-p)
       (org-inlinetask-in-task-p))
     ;; Inline task: Don't insert a new heading.
     (org-return t))

    ((org-at-table-p)
     (cond ((save-excursion
              (beginning-of-line)
              ;; See `org-table-next-field'.
              (cl-loop with end = (line-end-position)
                       for cell = (org-element-table-cell-parser)
                       always (equal (org-element-property :contents-begin cell)
                                     (org-element-property :contents-end cell))
                       while (re-search-forward "|" end t)))
            ;; Empty row: end the table.
            (delete-region (line-beginning-position) (line-end-position))
            (org-return t))
```

```
            (t
             ;; Non-empty row: call `org-return-indent'.
             (org-return t))))
     (t
      ;; All other cases: call `org-return-indent'.
      (org-return t)))))

(map!
 :after evil-org
 :map evil-org-mode-map
 :i [return] #'unpackaged/org-return-dwim)
```

Snippet Helper For snippets which want to depend on the `#+thing:` on the current line. This is mostly source blocks, and property args, so let's get fancy with them.

One-letter snippets are super-convenient, but for them to not be a pain everywhere else we'll need a nice condition function to use in yasnippet.

By having this function give slightly more than a simple `t` or `nil`, we can use in a second function to get the most popular language without explicit global header args.

```
(defun +yas/org-src-lang ()
  "Try to find the current language of the src/header at point.
     Return nil otherwise."
  (save-excursion
    (pcase
        (downcase
         (buffer-substring-no-properties
          (goto-char (line-beginning-position))
          (or (ignore-errors (1- (search-forward " " (line-end-position))))
              (1+ (point)))))
      ("#+property:"
       (when (re-search-forward "header-args:")
         (buffer-substring-no-properties
          (point)
          (or (and (forward-symbol 1) (point))
              (1+ (point))))))
      ("#+begin_src"
       (buffer-substring-no-properties
        (point)
        (or (and (forward-symbol 1) (point))
            (1+ (point)))))
      ("#+header:"
       (search-forward "#+begin_src")
       (+yas/org-src-lang))
      (_ nil))))

(defun +yas/org-src-header-p ()
```

```
  (or
   (looking-back "^#\\+property:[ \t]+header-args:.*" (line-beginning-position))
   (looking-back "^#\\+header:.*" (line-beginning-position))))

(defun +yas/org-last-src-lang ()
  (save-excursion
    (beginning-of-line)
    (when (search-backward "#+begin_src" nil t)
      (+yas/org-src-lang))))

(defun +yas/org-most-common-no-property-lang ()
  "Find the lang with the most source blocks that has no global header-args, else
  ↪ nil."
  (let (src-langs header-langs)
    (save-excursion
      (goto-char (point-min))
      (while (search-forward "#+begin_src" nil t)
        (push (+yas/org-src-lang) src-langs))
      (goto-char (point-min))
      (while (re-search-forward "#\\+property: +header-args" nil t)
        (push (+yas/org-src-lang) header-langs)))

    (setq src-langs
          (mapcar #'car
                  ;; sort alist by frequency (desc.)
                  (sort
                   ;; generate alist with form (value . frequency)
                   (cl-loop for (n . m) in (seq-group-by #'identity src-langs)
                            collect (cons n (length m)))
                   (lambda (a b) (> (cdr a) (cdr b))))))

    (car (cl-set-difference src-langs header-langs :test #'string=))))
```

Translate capital keywords (old) to lower case (new) Everyone used to use `#+CAPITAL` keywords. Then people realised that `#+lowercase` is actually both marginally easier and visually nicer, so now the capital version is just used in the manual.

> *Org is standardized on lower case. Uppercase is used in the manual as a poor man's bold, and supported for historical reasons.* — *Nicolas Goaziou on the Org ML*

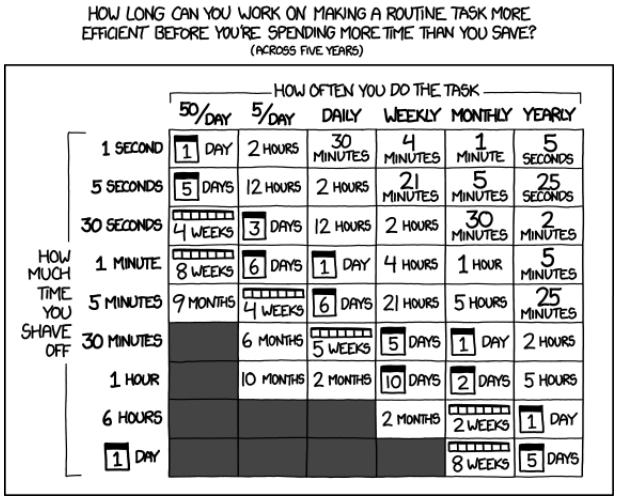
To avoid sometimes having to choose between the hassle out of updating old documents and using mixed syntax, I'll whip up a basic transcode-y function. It likely misses some edged cases, but should mostly work.

```
(defun org-syntax-convert-case-to-lower ()
  "Convert all #+KEYWORDS to #+keywords."
  (interactive)
  (save-excursion
```

```
    (goto-char (point-min))
    (let ((count 0)
          (case-fold-search nil))
      (while (re-search-forward "#\\+[A-Z_]+" nil t)
        (replace-match (downcase (match-string 0)) t)
        (setq count (1+ count)))
      (message "Replaced %d occurances" count))))
```

Extra links

xkcd Because xkcd is cool, let's make it as easy and fun as possible to insert them. Saving seconds adds up after all! (but only so much)



Is It Worth the Time? Don't forget the time you spend finding the chart to look up what you save. And the time spent reading this reminder about the time spent. And the time trying to figure out if either of those actually make sense. Remember, every second counts toward your life total, including these right now.

```
(org-link-set-parameters "xkcd"
                         :image-data-fun #'+org-xkcd-image-fn
                         :follow #'+org-xkcd-open-fn
                         :export #'+org-xkcd-export
                         :complete #'+org-xkcd-complete)

(defun +org-xkcd-open-fn (link)
  (+org-xkcd-image-fn nil link nil))

(defun +org-xkcd-image-fn (protocol link description)
  "Get image data for xkcd num LINK"
  (let* ((xkcd-info (+xkcd-fetch-info (string-to-number link)))
         (img (plist-get xkcd-info :img))
```

```
        (alt (plist-get xkcd-info :alt)))
    (message alt)
    (+org-image-file-data-fn protocol (xkcd-download img (string-to-number link))
    ↪  description)))

(defun +org-xkcd-export (num desc backend _com)
  "Convert xkcd to html/LaTeX form"
  (let* ((xkcd-info (+xkcd-fetch-info (string-to-number num)))
         (img (plist-get xkcd-info :img))
         (alt (plist-get xkcd-info :alt))
         (title (plist-get xkcd-info :title))
         (file (xkcd-download img (string-to-number num))))
    (cond ((org-export-derived-backend-p backend 'html)
           (format "<img class='invertible' src='%s' title=\"%s\" alt='%s'>" img
           ↪  (subst-char-in-string ?\" ?¿ alt) title))
          ((org-export-derived-backend-p backend 'latex)
           (format "\\begin{figure}[!htb]
                \\centering
                \\includegraphics[scale=0.4]{%s}%s
              \\end{figure}" file (if (equal desc (format "xkcd:%s" num)) ""
                     (format "\n  \\caption*{\\label{xkcd:%s} %s}"
                             num
                             (or desc
                                 (format "\\textbf{%s} %s" title alt))))))
          (t (format "https://xkcd.com/%s" num)))))

(defun +org-xkcd-complete (&optional arg)
  "Complete xkcd using `+xkcd-stored-info'"
  (format "xkcd:%d" (+xkcd-select)))
```

Music First, we set up all the necessarily 'utility' functions.

```
(after! org
  (defvar org-music-player 'mpris
    "Music player type. Curretly only supports mpris.")
  (defvar org-music-mpris-player "Lollypop"
    "Name of the mpris player, used in the form org.gnome.MPRIS.")
  (defvar org-music-track-search-method 'beets
    "Method to find the track file from the link.")
  (defvar org-music-beets-db "~/Music/library.db"
    "Location of the beets DB, for when using beets as the
    ↪  `org-music-track-search-method'")
  (defvar org-music-folder "~/Music/"
    "Location of your music folder, for when using file as the
    ↪  `org-music-track-search-method'")
  (defvar org-music-recognised-extensions '("flac" "mp4" "m4a" "aiff" "wav" "ogg"
  ↪  "aiff")
    "When searching for files in `org-music-track-search-method', recognise these
    ↪  extensions as audio files.")
```

```
(defun org-music-get-link (full &optional include-time)
  "Generate link string for currently playing track, optionally including a
  ↪  time-stamp"
  (pcase org-music-player ;; NOTE this could do with better generalisation
    ('mpris (let* ((track-metadata
                    (org-music-mpris-get-property "Metadata"))
                   (album-artist (caar (cadr (assoc "xesam:albumArtist"
                   ↪  track-metadata))))
                   (artist (if (or (equal album-artist "")
                                   (s-contains-p "various" album-artist t))
                               (caar (cadr (assoc "xesam:artist" track-metadata)))
                             album-artist))
                   (track (car (cadr (assoc "xesam:title" track-metadata))))
                   (start-time (when include-time
                                 (/ (org-music-mpris-get-property "Position")
                                 ↪  1000000))))
              (if full
                  (format "[[music:%s][%s by %s]]" (org-music-format-link artist
                  ↪  track start-time) track artist)
                (org-music-format-link artist track start-time))))
    (_ (user-error! "The specified music player: %s is not supported"
    ↪  org-music-player))))

(defun org-music-format-link (artist track &optional start-time end-time)
  (let ((artist (replace-regexp-in-string ":" "\\:" artist))
        (track (replace-regexp-in-string ":" "\\:" track)))
    (concat artist ":" track
            (cond ((and start-time end-time)
                   (format "::%s-%s"
                           (org-music-seconds-to-time start-time)
                           (org-music-seconds-to-time end-time)))
                  (start-time
                   (format "::%s"
                           (org-music-seconds-to-time start-time))))))))

(defun org-music-parse-link (link)
  (let* ((link-dc (->> link
                       (replace-regexp-in-string "\\([^\\\\]\\)\\\\:"
                       ↪  "\\1#COLON#")
                       (replace-regexp-in-string "\\(::[a-z0-9]*[0-9]\\)\\'"
                       ↪  "\\1s")))
         (link-components (mapcar (lambda (lc) (replace-regexp-in-string "#COLON#"
         ↪  ":" lc))
                                  (s-split ":" link-dc)))
         (artist (nth 0 link-components))
         (track (nth 1 link-components))
         (durations (when (and (> (length link-components) 3)
                               (equal (nth 2 link-components) ""))
                      (s-split "-" (nth 3 link-components))))
         (start-time (when durations
                       (org-music-time-to-seconds (car durations))))
         (end-time (when (cdr durations)
                     (org-music-time-to-seconds (cadr durations)))))
```

```
    (list artist track start-time end-time)))

(defun org-music-seconds-to-time (seconds)
  "Convert a number of seconds to a nice human duration, e.g. 5m21s.
     This action is reversed by `org-music-time-to-seconds'."
  (if (< seconds 60)
      (format "%ss" seconds)
    (if (< seconds 3600)
        (format "%sm%ss" (/ seconds 60) (% seconds 60))
      (format "%sh%sm%ss" (/ seconds 3600) (/ (% seconds 3600) 60) (% seconds
      ↪  60)))))

(defun org-music-time-to-seconds (time-str)
  "Get the number of seconds in a string produced by
  ↪  `org-music-seconds-to-time'."
  (let* ((time-components (reverse (s-split "[a-z]" time-str)))
         (seconds (string-to-number (nth 1 time-components)))
         (minutes (when (> (length time-components) 2)
                    (string-to-number (nth 2 time-components))))
         (hours (when (> (length time-components) 3)
                  (string-to-number (nth 3 time-components)))))
    (+ (* 3600 (or hours 0)) (* 60 (or minutes 0)) seconds)))

(defun org-music-play-track (artist title &optional start-time end-time)
  "Play the track specified by ARTIST and TITLE, optionally skipping to START-TIME
  ↪  in, stopping at END-TIME."
  (if-let ((file (org-music-find-track-file artist title)))
      (pcase org-music-player
        ('mpris (org-music-mpris-play file start-time end-time))
        (_ (user-error! "The specified music player: %s is not supported"
        ↪  org-music-player)))
    (user-error! "Could not find the track '%s' by '%s'" title artist)))

(add-transient-hook! #'org-music-play-track
  (require 'dbus))

(defun org-music-mpris-play (file &optional start-time end-time)
  (let ((uri (url-encode-url (rng-file-name-uri file))))
    (org-music-mpris-call-method "OpenUri" uri)
    (let ((track-id (caadr (assoc "mpris:trackid"
                                  (org-music-mpris-get-property "Metadata")))))
      (when start-time
        (org-music-mpris-call-method "SetPosition" :object-path track-id
                                     :int64 (round (* start-time 1000000))))
      (when end-time
        (org-music-mpris-stop-at-time uri end-time)))))

(defun orgb3-music-mpris-stop-at-time (url end-time)
  "Check that url is playing, and if it is stop it at END-TIME."
  (when (equal url (caadr (assoc "xesam:url" (org-music-mpris-get-property
  ↪  "Metadata"))))
    (let* ((time-current (/ (/ (org-music-mpris-get-property "Position") 10000)
    ↪  100.0))
```

```
            (time-delta (- end-time time-current)))
       (message "%s" time-delta)
       (if (< time-delta 0)
           (org-music-mpris-call-method "Pause")
         (if (< time-delta 6)
             (run-at-time (max 0.001 (* 0.9 time-delta)) nil
             ↪  #'org-music-mpris-stop-at-time url end-time)
           (run-at-time 5 nil #'org-music-mpris-stop-at-time url end-time))))))

(defun org-music-mpris-get-property (property)
  "Return the value of org.mpris.MediaPlayer2.Player.PROPERTY."
  (dbus-get-property :session (concat "org.gnome." org-music-mpris-player)
                     "/org/mpris/MediaPlayer2" "org.mpris.MediaPlayer2.Player"
                     property))

(defun org-music-mpris-call-method (property &rest args)
  "Call org.mpris.MediaPlayer2.Player.PROPERTY with ARGS, returning the result."
  (apply #'dbus-call-method :session (concat "org.gnome." org-music-mpris-player)
         "/org/mpris/MediaPlayer2" "org.mpris.MediaPlayer2.Player"
         property args))

(defun org-music-guess-mpris-player ()
  (when-let ((players
              (-filter (lambda (interface)
                         (s-contains-p "org.mpris.MediaPlayer2" interface))
                       (dbus-call-method :session
                                         dbus-service-dbus
                                         dbus-path-dbus
                                         dbus-interface-dbus
                                         "ListNames"))))
    (replace-regexp-in-string "org\\.mpris\\.MediaPlayer2\\." "" (car players))))

(when (eq org-music-player 'mpris)
  (unless org-music-mpris-player
    (setq org-music-mpris-player (org-music-guess-mpris-player))))

(defun org-music-find-track-file (artist title)
  "Try to find the file for TRACK by ARTIST, using
  ↪  `org-music-track-search-method', returning nil if nothing could be found."
  (pcase org-music-track-search-method
    ('file (org-music-find-file artist title))
    ('beets (org-music-beets-find-file artist title))
    (_ (user-error! "The specified music search method: %s is not supported"
    ↪  org-music-track-search-method))))

(defun org-music-beets-find-file (artist title)
  "Find the file corresponding to a given artist and title."
  (let* ((artist-escaped (replace-regexp-in-string "\"" "\\\"" artist))
         (title-escaped (replace-regexp-in-string "\"" "\\\"" title))
         (file
          (or
           (shell-command-to-string
            (format
```

```
              "sqlite3 '%s' \"SELECT path FROM items WHERE albumartist IS '%s' AND
              ↪  title IS '%s' LIMIT 1 COLLATE NOCASE\""
              (expand-file-name org-music-beets-db) artist-escaped title-escaped))
            (shell-command-to-string
             (format
              "sqlite3 '%s' \"SELECT path FROM items WHERE artist IS '%s' AND title
              ↪  IS '%s' LIMIT 1 COLLATE NOCASE\""
              (expand-file-name org-music-beets-db) artist-escaped
              ↪  title-escaped)))))
    (if (> (length file) 0)
        (substring file 0 -1)
      )))

(defun org-music-find-file (artist title)
  "Try to find a file in `org-music-folder' which contains TITLE, looking first
  ↪  in ./ARTIST if possible."
  (when-let* ((music-folder (expand-file-name org-music-folder))
              (search-folders (or
                               (-filter ; look for folders which contain ARTIST
                                (lambda (file-or-folder)
                                  (and
                                   (s-contains-p artist (file-name-base
                                   ↪  file-or-folder) t)
                                   (file-directory-p file-or-folder)))
                                (directory-files music-folder t))
                               (list music-folder)))
              (extension-regex (format "\\.\\(?:%s\\)\\'" (s-join "\\|"
              ↪  org-music-recognised-extensions)))
              (tracks (-filter
                       (lambda (file)
                         (s-contains-p title (file-name-base file) t))
                       (-flatten (mapcar (lambda (dir)
                                           (directory-files-recursively dir
                                           ↪  extension-regex))
                                         search-folders)))))
    (when (> (length tracks) 1)
      (message "Warning: multiple matches for %s by %s found" title artist))
    (car tracks))))
```

Then we integrate this nicely with org-mode

```
(after! org
  (org-link-set-parameters "music"
                           :follow #'org-music-open-fn
                           :export #'org-music-export-text)

  (org-link-set-parameters "Music" ;; like music, but visually fancier
                           ;; FIXME this should work as far as I can tell
                           ;; :image-data-fun #'org-music-image-fn
                           :follow #'org-music-open-fn
                           :export #'org-music-fancy-export)
```

```
(defun org-music-open-fn (link)
  (apply #'org-music-play-track (org-music-parse-link link)))

(defun org-music-insert-current-track (&optional include-time)
  "Insert link to currest track, including a timestamp when the universal argument
  ↪  is supplied."
  (interactive "P")
  (pp include-time)
  (insert (org-music-get-link t include-time)))

(defun org-music-export-text (path desc backend _com &optional newline)
  (let* ((track-info (org-music-parse-link path))
         (artist (nth 0 track-info))
         (track (nth 1 track-info))
         (start-time (nth 2 track-info))
         (end-time (nth 3 track-info))
         (emphasise (cond ((org-export-derived-backend-p backend 'html)
                           (lambda (s) (format "<span style=\"font-style:
                           ↪  italic\">%s</span>" s)))
                          ((org-export-derived-backend-p backend 'latex)
                           (lambda (s) (format "\\emph{%s}" s)))
                          (t (lambda (s) s)))))
    (or desc
        (concat
         (cond ((and start-time end-time)
                (format "%s to %s seconds of%s" start-time end-time (or newline "
                ↪  ")))
               (start-time
                (format "%s seconds into%s" start-time (or newline " "))))
         (funcall emphasise track)
         (or newline " ")
         "by "
         artist))))

(defun org-music-cover-image (track-file)
  "Try to find a cover image for the track in the given location"
  (car (-filter (lambda (file)
                  (-contains-p '("png" "jpg" "jpeg") (file-name-extension file)))
                (directory-files (file-name-directory track-file) t "cover"))))

(defun org-music-image-fn (_protocol link _description)
  (when-let* ((track-data (org-music-parse-link link))
              (cover-file (org-music-cover-image
                           (org-music-find-track-file
                            (nth 0 track-data) (nth 1 track-data)))))
    (with-temp-buffer
      (set-buffer-multibyte nil)
      (setq buffer-file-coding-system 'binary)
      (insert-file-contents-literally cover-file)
      (buffer-substring-no-properties (point-min) (point-max)))))

(defun org-music-fancy-export (path desc backend _com)
```

```
    (let* ((track-data (org-music-parse-link path))
           (file (org-music-find-track-file
                  (nth 0 track-data) (nth 1 track-data)))
           (cover-img (org-music-cover-image file))
           (newline-str (cond ((org-export-derived-backend-p backend 'html) "<br>")
                              ((org-export-derived-backend-p backend 'latex)
                              ↪  "\\newline ")
                              (t " ")))
           (text (org-music-export-text path nil backend nil newline-str)))
      (cond ((org-export-derived-backend-p backend 'html)
             (format "<div class='music-track'>
                     <img src='%s'> <span>%s</span>
                 </div>" cover-img text)
             )
            ((org-export-derived-backend-p backend 'latex)
             (format
                 "\\begin{tabular}{@{\\hspace{0.3\\columnwidth}}r@{\\hspace{0.1\\columnwidth}}p{0.4\\col
                   \\includegraphics[height=6em]{%s} &
                 \\vspace{-0.12\\columnwidth}%s
                 \\end{tabular}" cover-img text))
            (t text)))))
```

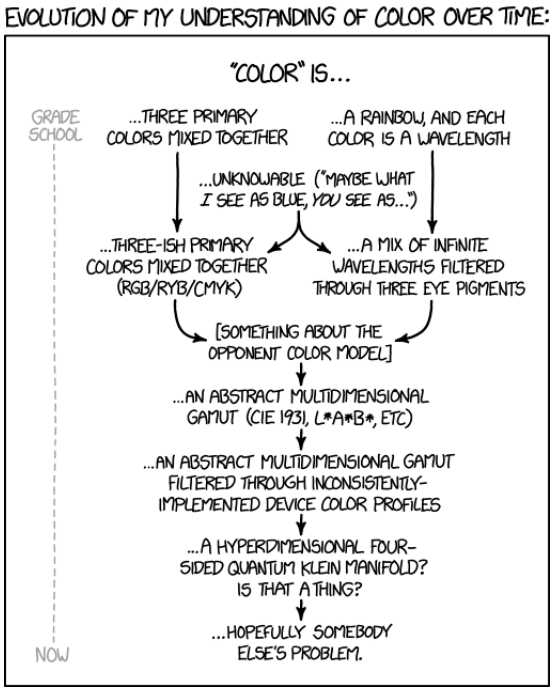
YouTube The `[[yt:...]]` links preview nicely, but don't export nicely. Thankfully, we can fix that.

```
(org-link-set-parameters "yt" :export #'+org-export-yt)
(defun +org-export-yt (path desc backend _com)
  (cond ((org-export-derived-backend-p backend 'html)
         (format "<iframe width='440' \
             height='335' \
             src='https://www.youtube.com/embed/%s' \
             frameborder='0' \
             allowfullscreen>%s</iframe>" path (or "" desc)))
        ((org-export-derived-backend-p backend 'latex)
         (format "\\href{https://youtu.be/%s}{%s}" path (or desc "youtube")))
        (t (format "https://youtu.be/%s" path))))
```

6.3.3 Visuals

Here I try to do two things: improve the styling of the various documents, via font changes etc, and also propagate colours from the current theme.

In editor



Color Models What if what *I* see as blue, *you* see as a slightly different blue because you're using Chrome instead of Firefox and despite a decade of messing with profiles we STILL can't get this right somehow.

Font Display Mixed pitch is great. As is `+org-pretty-mode`, let's use them.

```
(add-hook! 'org-mode-hook #'+org-pretty-mode #'mixed-pitch-mode)
```

Earlier I loaded the `org-pretty-table` package, let's enable it everywhere!

```
(setq global-org-pretty-table-mode t)
```

Let's make headings a bit bigger

```
(custom-set-faces!
  '(outline-1 :weight extra-bold :height 1.25)
  '(outline-2 :weight bold :height 1.15)
  '(outline-3 :weight bold :height 1.12)
  '(outline-4 :weight semi-bold :height 1.09)
  '(outline-5 :weight semi-bold :height 1.06)
  '(outline-6 :weight semi-bold :height 1.03)
  '(outline-8 :weight semi-bold)
  '(outline-9 :weight semi-bold))
```

And the same with the title.

```
(custom-set-faces!
  '(org-document-title :height 1.2))
```

It seems reasonable to have deadlines in the error face when they're passed.

```
(setq org-agenda-deadline-faces
      '((1.001 . error)
        (1.0 . org-warning)
        (0.5 . org-upcoming-deadline)
        (0.0 . org-upcoming-distant-deadline)))
```

We can then have quote blocks stand out a bit more by making them *italic*.

```
(setq org-fontify-quote-and-verse-blocks t)
```

While `org-hide-emphasis-markers` is very nice, it can sometimes make edits which occur at the border a bit more fiddley. We can improve this situation without sacrificing visual amenities with the `org-appear` package.

```
(use-package! org-appear
  :hook (org-mode . org-appear-mode)
  :config
  (setq org-appear-autoemphasis t
        org-appear-autosubmarkers t
        org-appear-autolinks nil)
  ;; for proper first-time setup, `org-appear--set-fragments'
  ;; needs to be run after other hooks have acted.
  (run-at-time nil nil #'org-appear--set-fragments))
```

Symbols It's also nice to change the character used for collapsed items (by default ¿), I think ¿ is better for indicating 'collapsed section'. and add an extra `org-bullet` to the default list of four. I've also added some fun alternatives, just commented out.

```
;; (use-package org-pretty-tags
;; :config
;;  (setq org-pretty-tags-surrogate-strings
;;        `(("uni"        . ,(all-the-icons-faicon   "graduation-cap" :face
↪  'all-the-icons-purple  :v-adjust 0.01))
;;          ("ucc"        . ,(all-the-icons-material "computer"       :face
↪  'all-the-icons-silver  :v-adjust 0.01))
;;          ("assignment" . ,(all-the-icons-material "library_books"  :face
↪  'all-the-icons-orange  :v-adjust 0.01))
```

```
;;          ("test"       . ,(all-the-icons-material "timer"          :face
↪  'all-the-icons-red     :v-adjust 0.01))
;;          ("lecture"    . ,(all-the-icons-fileicon "keynote"        :face
↪  'all-the-icons-orange  :v-adjust 0.01))
;;          ("email"      . ,(all-the-icons-faicon   "envelope"       :face
↪  'all-the-icons-blue    :v-adjust 0.01))
;;          ("read"       . ,(all-the-icons-octicon  "book"           :face
↪  'all-the-icons-lblue   :v-adjust 0.01))
;;          ("article"    . ,(all-the-icons-octicon  "file-text"      :face
↪  'all-the-icons-yellow  :v-adjust 0.01))
;;          ("web"        . ,(all-the-icons-faicon   "globe"          :face
↪  'all-the-icons-green   :v-adjust 0.01))
;;          ("info"       . ,(all-the-icons-faicon   "info-circle"    :face
↪  'all-the-icons-blue    :v-adjust 0.01))
;;          ("issue"      . ,(all-the-icons-faicon   "bug"            :face
↪  'all-the-icons-red     :v-adjust 0.01))
;;          ("someday"    . ,(all-the-icons-faicon   "calendar-o"     :face
↪  'all-the-icons-cyan    :v-adjust 0.01))
;;          ("idea"       . ,(all-the-icons-octicon  "light-bulb"     :face
↪  'all-the-icons-yellow  :v-adjust 0.01))
;;          ("emacs"      . ,(all-the-icons-fileicon "emacs"          :face
↪  'all-the-icons-lpurple :v-adjust 0.01))))
;;  (org-pretty-tags-global-mode))

(after! org-superstar
  (setq org-superstar-headline-bullets-list '("¿" "¿" "¿" "¿" "¿" "¿" "¿" "¿")
        ;; org-superstar-headline-bullets-list '("¿" "¿" "¿" "¿" "¿" "¿" "¿" "¿" "¿"
        ↪  "¿")
        org-superstar-prettify-item-bullets t ))

(setq org-ellipsis " ¿ "
      org-hide-leading-stars t
      org-priority-highest ?A
      org-priority-lowest ?E
      org-priority-faces
      '((?A . 'all-the-icons-red)
        (?B . 'all-the-icons-orange)
        (?C . 'all-the-icons-yellow)
        (?D . 'all-the-icons-green)
        (?E . 'all-the-icons-blue)))
```

It's also nice to make use of the Unicode characters for check boxes, and other commands.

```
(appendq! +ligatures-extra-symbols
          `(:checkbox      "¿"
            :pending       "¿"
            :checkedbox    "¿"
            :list_property "¿"
            :em_dash       "¿"
            :ellipses      "¿"
            :title         "¿"
            :subtitle      "¿"
```

```
            :author        "¿"
            :date          "¿"
            :property      "¿"
            :options       "¿"
            :latex_class   "¿"
            :latex_header  "¿"
            :beamer_header "¿"
            :attr_latex    "¿"
            :attr_html     "¿"
            :begin_quote   "¿"
            :end_quote     "¿"
            :caption       "¿"
            :header        "¿"
            :results       "¿"
            :begin_export  "¿"
            :end_export    "¿"
            :properties    "¿"
            :end           "¿"
            :priority_a   ,(propertize "¿" 'face 'all-the-icons-red)
            :priority_b   ,(propertize "¿" 'face 'all-the-icons-orange)
            :priority_c   ,(propertize "¿" 'face 'all-the-icons-yellow)
            :priority_d   ,(propertize "¿" 'face 'all-the-icons-green)
            :priority_e   ,(propertize "¿" 'face 'all-the-icons-blue)))
(set-ligatures! 'org-mode
  :merge t
  :checkbox      "[ ]"
  :pending       "[-]"
  :checkedbox    "[X]"
  :list_property "::"
  :em_dash       "---"
  :ellipsis      "..."
  :title         "#+title:"
  :subtitle      "#+subtitle:"
  :author        "#+author:"
  :date          "#+date:"
  :property      "#+property:"
  :options       "#+options:"
  :latex_class   "#+latex_class:"
  :latex_header  "#+latex_header:"
  :beamer_header "#+beamer_header:"
  :attr_latex    "#+attr_latex:"
  :attr_html     "#+attr_latex:"
  :begin_quote   "#+begin_quote"
  :end_quote     "#+end_quote"
  :caption       "#+caption:"
  :header        "#+header:"
  :begin_export  "#+begin_export"
  :end_export    "#+end_export"
  :results       "#+RESULTS:"
  :property      ":PROPERTIES:"
  :end           ":END:"
  :priority_a    "[#A]"
  :priority_b    "[#B]"
```

```
  :priority_c    "[#C]"
  :priority_d    "[#D]"
  :priority_e    "[#E]")
(plist-put +ligatures-extra-symbols :name "¿")
```

LaTeX Fragments First off, we want those fragments to look good.

```
(setq org-highlight-latex-and-related '(native script entities))
```

What's better than syntax-highlighted LaTeX is *rendered* LaTeX though, and we can have this be performed automatically with `org-fragtog`.

```
(use-package! org-fragtog
  :hook (org-mode . org-fragtog-mode))
```

It's nice to customise the look of LaTeX fragments so they fit better in the text — like this $\sqrt{\beta^2+3} - \sum_{\phi=1}^{\infty} \frac{x^{\phi-1}}{\Gamma(a)}$. Let's start by adding a sans font.

```
(setq org-format-latex-header "\\documentclass{article}
    \\usepackage[usenames]{color}
    \\usepackage[T1]{fontenc}
    \\usepackage{booktabs}
    \\pagestyle{empty}             % do not remove
    % The settings below are copied from fullpage.sty
    \\setlength{\\textwidth}{\\paperwidth}
    \\addtolength{\\textwidth}{-3cm}
    \\setlength{\\oddsidemargin}{1.5cm}
    \\addtolength{\\oddsidemargin}{-2.54cm}
    \\setlength{\\evensidemargin}{\\oddsidemargin}
    \\setlength{\\textheight}{\\paperheight}
    \\addtolength{\\textheight}{-\\headheight}
    \\addtolength{\\textheight}{-\\headsep}
    \\addtolength{\\textheight}{-\\footskip}
    \\addtolength{\\textheight}{-3cm}
    \\setlength{\\topmargin}{1.5cm}
    \\addtolength{\\topmargin}{-2.54cm}
    % my custom stuff
    \\usepackage[nofont,plaindd]{bmc-maths}
    \\usepackage{arev}
    ")
```

We can either render from a `dvi` or `pdf` file, so let's benchmark `latex` and `pdflatex`.

`latex` time	`pdflatex` time
135 ± 2 ms	215 ± 3 ms

On the rendering side, there are two `.dvi`-to-image converters which I am interested in: `dvipng` and `dvisvgm`. Then with the a `.pdf` we have `pdf2svg`. For inline preview we care about speed, while for exporting we care about file size and prefer a vector graphic.

Using the above latex expression and benchmarking lead to the following results:

`dvipng` time	`dvisvgm` time	`pdf2svg` time
89 ± 2 ms	178 ± 2 ms	12 ± 2 ms

Now let's combine this to see what's best

Tool chain	Total time	Resultant file size
`latex` + `dvipng`	226 ± 2 ms	7 KiB
`latex` + `dvisvgm`	392 ± 4 ms	8 KiB
`pdflatex` + `pdf2svg`	230 ± 2 ms	16 KiB

So, let's use `dvipng` for previewing LaTeX fragments in-Emacs, but `dvisvgm` for LaTeX Rendering. *Unfortunately: it seems that svg sizing is annoying ATM, so let's actually not do this right now.*

As well as having a sans font, there are a few other tweaks which can make them look better. Namely making sure that the colours switch when the theme does.

```
;; make background of fragments transparent
;; (let ((dvipng--plist (alist-get 'dvipng org-preview-latex-process-alist)))
;;   (plist-put dvipng--plist :use-xcolor t)
;;   (plist-put dvipng--plist :image-converter '("dvipng -D %D -bg 'transparent' -T
↪  tight -o %O %f")))
(add-hook! 'doom-load-theme-hook
  (defun +org-refresh-latex-background ()
    (plist-put! org-format-latex-options
                :background
                (face-attribute (or (cadr (assq 'default face-remapping-alist))
                                    'default)
                                :background nil t))))
```

It'd be nice to make `mhchem` equations able to be rendered. NB: This doesn't work at the moment.

```
(add-to-list 'org-latex-regexps '("\\ce" "^\\\\ce{\\(?:[^\000{}]\\|{[^\000}]+?}\\)}"
↪  0 nil))
```

Stolen from scimax (semi-working right now) I want fragment justification

```
(defun scimax-org-latex-fragment-justify (justification)
  "Justify the latex fragment at point with JUSTIFICATION.
      JUSTIFICATION is a symbol for 'left, 'center or 'right."
  (interactive
   (list (intern-soft
          (completing-read "Justification (left): " '(left center right)
                           nil t nil nil 'left))))
  (let* ((ov (ov-at))
         (beg (ov-beg ov))
         (end (ov-end ov))
         (shift (- beg (line-beginning-position)))
         (img (overlay-get ov 'display))
         (img (and (and img (consp img) (eq (car img) 'image)
                        (image-type-available-p (plist-get (cdr img) :type)))
                   img))
         space-left offset)
    (when (and img
               ;; This means the equation is at the start of the line
               (= beg (line-beginning-position))
               (or
                (string= "" (s-trim (buffer-substring end (line-end-position))))
                (eq 'latex-environment (car (org-element-context)))))
      (setq space-left (- (window-max-chars-per-line) (car (image-size img)))
            offset (floor (cond
                           ((eq justification 'center)
                            (- (/ space-left 2) shift))
                           ((eq justification 'right)
                            (- space-left shift))
                           (t
                            0))))
      (when (>= offset 0)
        (overlay-put ov 'before-string (make-string offset ?\ ))))))

(defun scimax-org-latex-fragment-justify-advice (beg end image imagetype)
  "After advice function to justify fragments."
  (scimax-org-latex-fragment-justify (or (plist-get org-format-latex-options
  ↪  :justify) 'left)))


(defun scimax-toggle-latex-fragment-justification ()
  "Toggle if LaTeX fragment justification options can be used."
  (interactive)
  (if (not (get 'scimax-org-latex-fragment-justify-advice 'enabled))
      (progn
```

```
        (advice-add 'org--format-latex-make-overlay :after
        ↪  'scimax-org-latex-fragment-justify-advice)
        (put 'scimax-org-latex-fragment-justify-advice 'enabled t)
        (message "Latex fragment justification enabled"))
    (advice-remove 'org--format-latex-make-overlay
    ↪  'scimax-org-latex-fragment-justify-advice)
    (put 'scimax-org-latex-fragment-justify-advice 'enabled nil)
    (message "Latex fragment justification disabled")))
```

There's also this lovely equation numbering stuff I'll nick

```
;; Numbered equations all have (1) as the number for fragments with vanilla
;; org-mode. This code injects the correct numbers into the previews so they
;; look good.
(defun scimax-org-renumber-environment (orig-func &rest args)
  "A function to inject numbers in LaTeX fragment previews."
  (let ((results '())
        (counter -1)
        (numberp))
    (setq results (cl-loop for (begin . env) in
                           (org-element-map (org-element-parse-buffer)
                           ↪  'latex-environment
                             (lambda (env)
                               (cons
                                (org-element-property :begin env)
                                (org-element-property :value env))))
                           collect
                           (cond
                            ((and (string-match "\\\\begin{equation}" env)
                                  (not (string-match "\\\\tag{" env)))
                             (incf counter)
                             (cons begin counter))
                            ((string-match "\\\\begin{align}" env)
                             (prog2
                                 (incf counter)
                                 (cons begin counter)
                               (with-temp-buffer
                                 (insert env)
                                 (goto-char (point-min))
                                 ;; \\ is used for a new line. Each one leads to a
                                 ↪  number
                                 (incf counter (count-matches "\\\\$"))
                                 ;; unless there are nonumbers.
                                 (goto-char (point-min))
                                 (decf counter (count-matches "\\nonumber")))))
                            (t
                             (cons begin nil)))))

    (when (setq numberp (cdr (assoc (point) results)))
      (setf (car args)
            (concat
```

```
            (format "\\setcounter{equation}{%s}\n" numberp)
            (car args)))))

  (apply orig-func args))


(defun scimax-toggle-latex-equation-numbering ()
  "Toggle whether LaTeX fragments are numbered."
  (interactive)
  (if (not (get 'scimax-org-renumber-environment 'enabled))
      (progn
        (advice-add 'org-create-formula-image :around
        ↪  #'scimax-org-renumber-environment)
        (put 'scimax-org-renumber-environment 'enabled t)
        (message "Latex numbering enabled"))
    (advice-remove 'org-create-formula-image #'scimax-org-renumber-environment)
    (put 'scimax-org-renumber-environment 'enabled nil)
    (message "Latex numbering disabled.")))

(advice-add 'org-create-formula-image :around #'scimax-org-renumber-environment)
(put 'scimax-org-renumber-environment 'enabled t)
```

Org Plot We can use some of the variables in `org-plot` to use the current doom theme colours.

```
(after! org-plot
  (defun org-plot/generate-theme (_type)
    "Use the current Doom theme colours to generate a GnuPlot preamble."
```

```
    (format "
        fgt = \"textcolor rgb '%s'\" # foreground text
        fgat = \"textcolor rgb '%s'\" # foreground alt text
        fgl = \"linecolor rgb '%s'\" # foreground line
        fgal = \"linecolor rgb '%s'\" # foreground alt line
        # foreground colors
        set border lc rgb '%s'
        # change text colors of  tics
        set xtics @fgt
        set ytics @fgt
        # change text colors of labels
        set title @fgt
        set xlabel @fgt
        set ylabel @fgt
        # change a text color of key
        set key @fgt
        # line styles
        set linetype 1 lw 2 lc rgb '%s' # red
        set linetype 2 lw 2 lc rgb '%s' # blue
        set linetype 3 lw 2 lc rgb '%s' # green
        set linetype 4 lw 2 lc rgb '%s' # magenta
        set linetype 5 lw 2 lc rgb '%s' # orange
        set linetype 6 lw 2 lc rgb '%s' # yellow
        set linetype 7 lw 2 lc rgb '%s' # teal
        set linetype 8 lw 2 lc rgb '%s' # violet
        # palette
        set palette maxcolors 8
        set palette defined ( 0 '%s',\
        1 '%s',\
        2 '%s',\
        3 '%s',\
        4 '%s',\
        5 '%s',\
        6 '%s',\
        7 '%s' )
        "
            (doom-color 'fg)
            (doom-color 'fg-alt)
            (doom-color 'fg)
            (doom-color 'fg-alt)
            (doom-color 'fg)
            ;; colours
            (doom-color 'red)
            (doom-color 'blue)
            (doom-color 'green)
            (doom-color 'magenta)
            (doom-color 'orange)
            (doom-color 'yellow)
            (doom-color 'teal)
            (doom-color 'violet)
            ;; duplicated
            (doom-color 'red)
            (doom-color 'blue)
```

```
            (doom-color 'green)
            (doom-color 'magenta)
            (doom-color 'orange)
            (doom-color 'yellow)
            (doom-color 'teal)
            (doom-color 'violet)
            ))
  (defun org-plot/gnuplot-term-properties (_type)
    (format "background rgb '%s' size 1050,650"
            (doom-color 'bg)))
  (setq org-plot/gnuplot-script-preamble #'org-plot/generate-theme)
  (setq org-plot/gnuplot-term-extra #'org-plot/gnuplot-term-properties))
```

Exporting (general)

```
(setq org-export-headline-levels 5) ; I like nesting
```

I'm also going to make use of an item in `ox-extra` so that I can add an `:ignore:` tag to headings for the content to be kept, but the heading itself ignored (unlike `:noexport:` which ignored both heading and content). This is useful when I want to use headings to provide a structure for writing that doesn't appear in the final documents.

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

Exporting to HTML I want to tweak a whole bunch of things. While I'll want my tweaks almost all the time, occasionally I may want to test how something turns out using a more default config. With that in mind, a global minor mode seems like the most appropriate architecture to use.

```
(define-minor-mode org-fancy-html-export-mode
  "Toggle my fabulous org export tweaks. While this mode itself does a little bit,
      the vast majority of the change in behaviour comes from switch statements in:
       - `org-html-template-fancier'
       - `org-html--build-meta-info-extended'
       - `org-html-src-block-collapsable'
       - `org-html-block-collapsable'
       - `org-html-table-wrapped'
       - `org-html--format-toc-headline-colapseable'
       - `org-html--toc-text-stripped-leaves'
       - `org-export-html-headline-anchor'"
  :global t
  :init-value t
  (if org-fancy-html-export-mode
      (setq org-html-style-default org-html-style-fancy
```

```
            org-html-meta-tags #'org-html-meta-tags-fancy
            org-html-checkbox-type 'html-span)
    (setq org-html-style-default org-html-style-plain
          org-html-meta-tags #'org-html-meta-tags-default
          org-html-checkbox-type 'html)))
```

There are quite a few instances where I want to modify variables defined in `ox-html`, so we'll wrap the contents of this section in an `(after! ox-html ...)` block.

```
(after! ox-html
  (define-minor-mode org-fancy-html-export-mode
    "Toggle my fabulous org export tweaks. While this mode itself does a little bit,
          the vast majority of the change in behaviour comes from switch statements
        in:
           - `org-html-template-fancier'
           - `org-html--build-meta-info-extended'
           - `org-html-src-block-collapsable'
           - `org-html-block-collapsable'
           - `org-html-table-wrapped'
           - `org-html--format-toc-headline-colapseable'
           - `org-html--toc-text-stripped-leaves'
           - `org-export-html-headline-anchor'"
    :global t
    :init-value t
    (if org-fancy-html-export-mode
        (setq org-html-style-default org-html-style-fancy
              org-html-meta-tags #'org-html-meta-tags-fancy
              org-html-checkbox-type 'html-span)
      (setq org-html-style-default org-html-style-plain
            org-html-meta-tags #'org-html-meta-tags-default
            org-html-checkbox-type 'html)))
  (defadvice! org-html-template-fancier (orig-fn contents info)
    "Return complete document string after HTML conversion.
          CONTENTS is the transcoded contents string.  INFO is a plist
          holding export options. Adds a few extra things to the body
          compared to the default implementation."
    :around #'org-html-template
    (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪  +org-msg-currently-exporting))
        (funcall orig-fn contents info)
      (concat
       (when (and (not (org-html-html5-p info)) (org-html-xhtml-p info))
         (let* ((xml-declaration (plist-get info :html-xml-declaration))
                (decl (or (and (stringp xml-declaration) xml-declaration)
                          (cdr (assoc (plist-get info :html-extension)
                                      xml-declaration))
                          (cdr (assoc "html" xml-declaration))
                          "")))
           (when (not (or (not decl) (string= "" decl)))
             (format "%s\n"
                     (format decl
```

```
                        (or (and org-html-coding-system
                                 (fboundp 'coding-system-get)
                                 (coding-system-get org-html-coding-system
                                 ↪  'mime-charset))
                            "iso-8859-1"))))))
(org-html-doctype info)
"\n"
(concat "<html"
        (cond ((org-html-xhtml-p info)
               (format
                " xmlns=\"http://www.w3.org/1999/xhtml\" lang=\"%s\"
                ↪  xml:lang=\"%s\""
                (plist-get info :language) (plist-get info :language)))
              ((org-html-html5-p info)
               (format " lang=\"%s\"" (plist-get info :language))))
        ">\n")
"<head>\n"
(org-html--build-meta-info info)
(org-html--build-head info)
(org-html--build-mathjax-config info)
"</head>\n"
"<body>\n<input type='checkbox' id='theme-switch'><div id='page'><label
↪  id='switch-label' for='theme-switch'></label>"
(let ((link-up (org-trim (plist-get info :html-link-up)))
      (link-home (org-trim (plist-get info :html-link-home))))
  (unless (and (string= link-up "") (string= link-home ""))
    (format (plist-get info :html-home/up-format)
            (or link-up link-home)
            (or link-home link-up))))
;; Preamble.
(org-html--build-pre/postamble 'preamble info)
;; Document contents.
(let ((div (assq 'content (plist-get info :html-divs))))
  (format "<%s id=\"%s\">\n" (nth 1 div) (nth 2 div)))
;; Document title.
(when (plist-get info :with-title)
  (let ((title (and (plist-get info :with-title)
                    (plist-get info :title)))
        (subtitle (plist-get info :subtitle))
        (html5-fancy (org-html--html5-fancy-p info)))
    (when title
      (format
       "<div class='page-header'><div class='page-meta'>%s, %s</div><h1
       ↪  class=\"title\">%s%s</h1></div>\n"
       (org-export-data (plist-get info :date) info)
       (org-export-data (plist-get info :author) info)
       (org-export-data title info)
       (if subtitle
           (format
            (if html5-fancy
                "<p class=\"subtitle\">%s</p>\n"
              (concat "\n" (org-html-close-tag "br" nil info) "\n"
                      "<span class=\"subtitle\">%s</span>\n"))
```

```
                  (org-export-data subtitle info))
                "")))))
      contents
      (format "</%s>\n" (nth 1 (assq 'content (plist-get info :html-divs))))
      ;; Postamble.
      (org-html--build-pre/postamble 'postamble info)
      ;; Possibly use the Klipse library live code blocks.
      (when (plist-get info :html-klipsify-src)
        (concat "<script>" (plist-get info :html-klipse-selection-script)
                "</script><script src=\""
                org-html-klipse-js
                "\"></script><link rel=\"stylesheet\" type=\"text/css\" href=\""
                org-html-klipse-css "\"/>"))
      ;; Closing document.
      "</div>\n</body>\n</html>")))
(defadvice! org-html-toc-linked (depth info &optional scope)
  "Build a table of contents.
        Just like `org-html-toc', except the header is a link to \"#\".
        DEPTH is an integer specifying the depth of the table.  INFO is
        a plist used as a communication channel.  Optional argument SCOPE
        is an element defining the scope of the table.  Return the table
        of contents as a string, or nil if it is empty."
  :override #'org-html-toc
  (let ((toc-entries
     (mapcar (lambda (headline)
           (cons (org-html--format-toc-headline headline info)
             (org-export-get-relative-level headline info)))
         (org-export-collect-headlines info depth scope))))
    (when toc-entries
      (let ((toc (concat "<div id=\"text-table-of-contents\">"
             (org-html--toc-text toc-entries)
             "</div>\n")))
    (if scope toc
      (let ((outer-tag (if (org-html--html5-fancy-p info)
                   "nav"
                 "div")))
        (concat (format "<%s id=\"table-of-contents\">\n" outer-tag)
            (let ((top-level (plist-get info :html-toplevel-hlevel)))
              (format "<h%d><a href=\"#\" style=\"color:inherit; text-decoration:
              ↪  none;\">%s</a></h%d>\n"
                  top-level
                  (org-html--translate "Table of Contents" info)
                  top-level))
            toc
            (format "</%s>\n" outer-tag)))))))))
(defun org-html-meta-tags-fancy (info)
  "Use the INFO plist to construct the meta tags, as described in
  ↪  `org-html-meta-tags'."
  (let ((title (org-html-plain-text
                (org-element-interpret-data (plist-get info :title)) info))
        (author (and (plist-get info :with-author)
                     (let ((auth (plist-get info :author)))
                       ;; Return raw Org syntax.
```

```
                        (and auth (org-html-plain-text
                                   (org-element-interpret-data auth) info))))))
     (list
      (when (org-string-nw-p author)
        (list "name" "author" author))
      (when (org-string-nw-p (plist-get info :description))
        (list "name" "description"
              (plist-get info :description)))
      '("name" "generator" "org mode")
      '("name" "theme-color" "#77aa99")
      '("property" "og:type" "article")
      (list "property" "og:title" title)
      (let ((subtitle (org-export-data (plist-get info :subtitle) info)))
        (when (org-string-nw-p subtitle)
          (list "property" "og:description" subtitle)))
      '("property" "og:image" "https://tecosaur.com/resources/org/nib.png")
      '("property" "og:image:type" "image/png")
      '("property" "og:image:width" "200")
      '("property" "og:image:height" "200")
      '("property" "og:image:alt" "Green fountain pen nib")
      (when (org-string-nw-p author)
        (list "property" "og:article:author:first_name" (car (s-split-up-to " "
        ↪  author 2))))
      (when (and (org-string-nw-p author) (s-contains-p " " author))
        (list "property" "og:article:author:last_name" (cadr (s-split-up-to " "
        ↪  author 2))))
      (list "property" "og:article:published_time" (format-time-string
      ↪  "%FT%T%z")))))

(unless (functionp #'org-html-meta-tags-default)
  (defalias 'org-html-meta-tags-default #'ignore))
(setq org-html-meta-tags #'org-html-meta-tags-fancy)
(setq org-html-style-plain org-html-style-default
      org-html-htmlize-output-type 'css
      org-html-doctype "html5"
      org-html-html5-fancy t)

(defun org-html-reload-fancy-style ()
  (interactive)
  (setq org-html-style-fancy
        (concat (f-read-text (expand-file-name "misc/org-export-header.html"
        ↪  doom-private-dir))
                "<script>\n"
                (f-read-text (expand-file-name "misc/org-css/main.js"
                ↪  doom-private-dir))
                "</script>\n<style>\n"
                (f-read-text (expand-file-name "misc/org-css/main.css"
                ↪  doom-private-dir))
                "</style>"))
  (when org-fancy-html-export-mode
    (setq org-html-style-default org-html-style-fancy)))
(org-html-reload-fancy-style)
(defvar org-html-export-collapsed nil)
```

```
(eval '(cl-pushnew '(:collapsed "COLLAPSED" "collapsed" org-html-export-collapsed
↪  t)
                   (org-export-backend-options (org-export-get-backend 'html))))
(add-to-list 'org-default-properties "EXPORT_COLLAPSED")
(defadvice! org-html-src-block-collapsable (orig-fn src-block contents info)
  "Wrap the usual <pre> block in a <details>"
  :around #'org-html-src-block
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪  +org-msg-currently-exporting))
      (funcall orig-fn src-block contents info)
    (let* ((properties (cadr src-block))
           (lang (mode-name-to-lang-name
                  (plist-get properties :language)))
           (name (plist-get properties :name))
           (ref (org-export-get-reference src-block info))
           (collapsed-p (member (or (org-export-read-attribute :attr_html
           ↪  src-block :collapsed)
                                    (plist-get info :collapsed))
                                '("y" "yes" "t" t "true" "all"))))
      (format
       "<details id='%s' class='code'%s><summary%s>%s</summary>
            <div class='gutter'>
            <a href='#%s'>#</a>
            <button title='Copy to clipboard'
           onclick='copyPreToClipbord(this)'>¿</button>\
            </div>
            %s
            </details>"
       ref
       (if collapsed-p "" " open")
       (if name " class='named'" "")
       (if (not name) (concat "<span class='lang'>" lang "</span>")
         (format "<span class='name'>%s</span><span class='lang'>%s</span>" name
         ↪  lang))
       ref
       (if name
           (replace-regexp-in-string (format "<pre\\( class=\"[^\"]+\"\\)?
           ↪  id=\"%s\">" ref) "<pre\\1>"
                                     (funcall orig-fn src-block contents info))
         (funcall orig-fn src-block contents info))))))

(defun mode-name-to-lang-name (mode)
  (or (cadr (assoc mode
                   '(("asymptote" "Asymptote")
                     ("awk" "Awk")
                     ("C" "C")
                     ("clojure" "Clojure")
                     ("css" "CSS")
                     ("D" "D")
                     ("ditaa" "ditaa")
                     ("dot" "Graphviz")
                     ("calc" "Emacs Calc")
                     ("emacs-lisp" "Emacs Lisp")
```

```
                    ("fortran" "Fortran")
                    ("gnuplot" "gnuplot")
                    ("haskell" "Haskell")
                    ("hledger" "hledger")
                    ("java" "Java")
                    ("js" "Javascript")
                    ("latex" "LaTeX")
                    ("ledger" "Ledger")
                    ("lisp" "Lisp")
                    ("lilypond" "Lilypond")
                    ("lua" "Lua")
                    ("matlab" "MATLAB")
                    ("mscgen" "Mscgen")
                    ("ocaml" "Objective Caml")
                    ("octave" "Octave")
                    ("org" "Org mode")
                    ("oz" "OZ")
                    ("plantuml" "Plantuml")
                    ("processing" "Processing.js")
                    ("python" "Python")
                    ("R" "R")
                    ("ruby" "Ruby")
                    ("sass" "Sass")
                    ("scheme" "Scheme")
                    ("screen" "Gnu Screen")
                    ("sed" "Sed")
                    ("sh" "shell")
                    ("sql" "SQL")
                    ("sqlite" "SQLite")
                    ("forth" "Forth")
                    ("io" "IO")
                    ("J" "J")
                    ("makefile" "Makefile")
                    ("maxima" "Maxima")
                    ("perl" "Perl")
                    ("picolisp" "Pico Lisp")
                    ("scala" "Scala")
                    ("shell" "Shell Script")
                    ("ebnf2ps" "ebfn2ps")
                    ("cpp" "C++")
                    ("abc" "ABC")
                    ("coq" "Coq")
                    ("groovy" "Groovy")
                    ("bash" "bash")
                    ("csh" "csh")
                    ("ash" "ash")
                    ("dash" "dash")
                    ("ksh" "ksh")
                    ("mksh" "mksh")
                    ("posh" "posh")
                    ("ada" "Ada")
                    ("asm" "Assembler")
                    ("caml" "Caml")
```

```
                        ("delphi" "Delphi")
                        ("html" "HTML")
                        ("idl" "IDL")
                        ("mercury" "Mercury")
                        ("metapost" "MetaPost")
                        ("modula-2" "Modula-2")
                        ("pascal" "Pascal")
                        ("ps" "PostScript")
                        ("prolog" "Prolog")
                        ("simula" "Simula")
                        ("tcl" "tcl")
                        ("tex" "LaTeX")
                        ("plain-tex" "TeX")
                        ("verilog" "Verilog")
                        ("vhdl" "VHDL")
                        ("xml" "XML")
                        ("nxml" "XML")
                        ("conf" "Configuration File"))))
      mode))
(defun org-html-block-collapsable (orig-fn block contents info)
  "Wrap the usual block in a <details>"
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪  +org-msg-currently-exporting))
      (funcall orig-fn block contents info)
    (let ((ref (org-export-get-reference block info))
          (type (pcase (car block)
                  ('property-drawer "Properties")))
          (collapsed-default (pcase (car block)
                               ('property-drawer t)
                               (_ nil)))
          (collapsed-value (org-export-read-attribute :attr_html block
          ↪  :collapsed))
          (collapsed-p (or (member (org-export-read-attribute :attr_html block
          ↪  :collapsed)
                                   '("y" "yes" "t" t "true"))
                           (member (plist-get info :collapsed) '("all")))))
      (format
       "<details id='%s' class='code'%s>
            <summary%s>%s</summary>
            <div class='gutter'>\
            <a href='#%s'>#</a>
            <button title='Copy to clipboard'
           onclick='copyPreToClipbord(this)'>¿</button>\
            </div>
            %s\n
            </details>"
       ref
       (if (or collapsed-p collapsed-default) "" " open")
       (if type " class='named'" "")
       (if type (format "<span class='type'>%s</span>" type) "")
       ref
       (funcall orig-fn block contents info)))))
```

```
(advice-add 'org-html-example-block   :around #'org-html-block-collapsable)
(advice-add 'org-html-fixed-width     :around #'org-html-block-collapsable)
(advice-add 'org-html-property-drawer :around #'org-html-block-collapsable)
(add-hook 'htmlize-before-hook #'highlight-numbers--turn-on)
(defadvice! org-html-table-wrapped (orig-fn table contents info)
  "Wrap the usual <table> in a <div>"
  :around #'org-html-table
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn table contents info)
    (let* ((name (plist-get (cadr table) :name))
           (ref (org-export-get-reference table info)))
      (format "<div id='%s' class='table'>
            <div class='gutter'><a href='#%s'>#</a></div>
            <div class='tabular'>
            %s
            </div>\
            </div>"
              ref ref
              (if name
                  (replace-regexp-in-string (format "<table id=\"%s\"" ref)
                  ↪  "<table"
                                            (funcall orig-fn table contents info))
                (funcall orig-fn table contents info))))))
(defadvice! org-html--format-toc-headline-colapseable (orig-fn headline info)
  "Add a label and checkbox to `org-html--format-toc-headline's usual output,
       to allow the TOC to be a collapseable tree."
  :around #'org-html--format-toc-headline
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn headline info)
    (let ((id (or (org-element-property :CUSTOM_ID headline)
                  (org-export-get-reference headline info))))
      (format "<input type='checkbox' id='toc--%s'/><label
      ↪  for='toc--%s'>%s</label>"
              id id (funcall orig-fn headline info)))))
(defadvice! org-html--toc-text-stripped-leaves (orig-fn toc-entries)
  "Remove label"
  :around #'org-html--toc-text
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn toc-entries)
    (replace-regexp-in-string "<input [^>]+><label [^>]+>\\(.+?\\)</label></li>"
    ↪  "\\1</li>"
                              (funcall orig-fn toc-entries))))
(setq org-html-text-markup-alist
      '((bold . "<b>%s</b>")
        (code . "<code>%s</code>")
        (italic . "<i>%s</i>")
        (strike-through . "<del>%s</del>")
        (underline . "<span class=\"underline\">%s</span>")
        (verbatim . "<kbd>%s</kbd>")))
(appendq! org-html-checkbox-types
```

```
            '((html-span
               (on . "<span class='checkbox'></span>")
               (off . "<span class='checkbox'></span>")
               (trans . "<span class='checkbox'></span>"))))
  (setq org-html-checkbox-type 'html-span)
  (defun org-export-html-headline-anchor (text backend info)
    (when (and (org-export-derived-backend-p backend 'html)
               org-fancy-html-export-mode)
      (unless (bound-and-true-p +org-msg-currently-exporting)
        (replace-regexp-in-string
         "<h\\([0-9]\\) id=\"\\([a-z0-9-]+\\)\">\\(.*[^ ]\\)<\\/h[0-9]>" ; this is
         ↪  quite restrictive, but due to `org-reference-contraction' I can do this
         "<h\\1 id=\"\\2\">\\3<a aria-hidden=\"true\" href=\"#\\2\">#</a> </h\\1>"
         text))))

  (add-to-list 'org-export-filter-headline-functions
               'org-export-html-headline-anchor)
  ;; (setq-default org-html-with-latex `dvisvgm)
  (setq org-html-mathjax-options
        '((path "https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-svg.js" )
          (scale "1")
          (autonumber "ams")
          (multlinewidth "85%")
          (tagindent ".8em")
          (tagside "right")))

  (setq org-html-mathjax-template
        "<script>
              MathJax = {
                chtml: {
                  scale: %SCALE
                },
                svg: {
                  scale: %SCALE,
                  fontCache: \"global\"
                },
                tex: {
                  tags: \"%AUTONUMBER\",
                  multlineWidth: \"%MULTLINEWIDTH\",
                  tagSide: \"%TAGSIDE\",
                  tagIndent: \"%TAGINDENT\"
                }
              };
              </script>
              <script id=\"MathJax-script\" async
                      src=\"%PATH\"></script>")
)
```

Extra header content We want to tack on a few more bits to the start of the body. Unfortunately, there doesn't seem to be any nice variable or hook, so we'll just override the relevant function.

This is done to allow me to add the date and author to the page header, implement a css-only light/dark theme toggle, and a sprinkle of Open Graph metadata.

```
(defadvice! org-html-template-fancier (orig-fn contents info)
  "Return complete document string after HTML conversion.
      CONTENTS is the transcoded contents string.  INFO is a plist
      holding export options. Adds a few extra things to the body
      compared to the default implementation."
  :around #'org-html-template
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪  +org-msg-currently-exporting))
      (funcall orig-fn contents info)
    (concat
     (when (and (not (org-html-html5-p info)) (org-html-xhtml-p info))
       (let* ((xml-declaration (plist-get info :html-xml-declaration))
              (decl (or (and (stringp xml-declaration) xml-declaration)
                        (cdr (assoc (plist-get info :html-extension)
                                    xml-declaration))
                        (cdr (assoc "html" xml-declaration))
                        "")))
         (when (not (or (not decl) (string= "" decl)))
           (format "%s\n"
                   (format decl
                           (or (and org-html-coding-system
                                    (fboundp 'coding-system-get)
                                    (coding-system-get org-html-coding-system
                                    ↪  'mime-charset))
                               "iso-8859-1"))))))
     (org-html-doctype info)
     "\n"
     (concat "<html"
             (cond ((org-html-xhtml-p info)
                    (format
                     " xmlns=\"http://www.w3.org/1999/xhtml\" lang=\"%s\"
                     ↪  xml:lang=\"%s\""
                     (plist-get info :language) (plist-get info :language)))
                   ((org-html-html5-p info)
                    (format " lang=\"%s\"" (plist-get info :language))))
             ">\n")
     "<head>\n"
     (org-html--build-meta-info info)
     (org-html--build-head info)
     (org-html--build-mathjax-config info)
     "</head>\n"
     "<body>\n<input type='checkbox' id='theme-switch'><div id='page'><label
     ↪  id='switch-label' for='theme-switch'></label>"
     (let ((link-up (org-trim (plist-get info :html-link-up)))
           (link-home (org-trim (plist-get info :html-link-home))))
       (unless (and (string= link-up "") (string= link-home ""))
         (format (plist-get info :html-home/up-format)
                 (or link-up link-home)
                 (or link-home link-up))))
```

```
    ;; Preamble.
    (org-html--build-pre/postamble 'preamble info)
    ;; Document contents.
    (let ((div (assq 'content (plist-get info :html-divs))))
      (format "<%s id=\"%s\">\n" (nth 1 div) (nth 2 div)))
    ;; Document title.
    (when (plist-get info :with-title)
      (let ((title (and (plist-get info :with-title)
                        (plist-get info :title)))
            (subtitle (plist-get info :subtitle))
            (html5-fancy (org-html--html5-fancy-p info)))
        (when title
          (format
           "<div class='page-header'><div class='page-meta'>%s, %s</div><h1
           ↪  class=\"title\">%s%s</h1></div>\n"
           (org-export-data (plist-get info :date) info)
           (org-export-data (plist-get info :author) info)
           (org-export-data title info)
           (if subtitle
               (format
                (if html5-fancy
                    "<p class=\"subtitle\">%s</p>\n"
                  (concat "\n" (org-html-close-tag "br" nil info) "\n"
                          "<span class=\"subtitle\">%s</span>\n"))
                (org-export-data subtitle info))
             ""))))))
    contents
    (format "</%s>\n" (nth 1 (assq 'content (plist-get info :html-divs))))
    ;; Postamble.
    (org-html--build-pre/postamble 'postamble info)
    ;; Possibly use the Klipse library live code blocks.
    (when (plist-get info :html-klipsify-src)
      (concat "<script>" (plist-get info :html-klipse-selection-script)
              "</script><script src=\""
              org-html-klipse-js
              "\"></script><link rel=\"stylesheet\" type=\"text/css\" href=\""
              org-html-klipse-css "\"/>"))
    ;; Closing document.
    "</div>\n</body>\n</html>")))
```

I think it would be nice if "Table of Contents" brought you back to the top of the page. Well, since we've done this much advising already...

```
(defadvice! org-html-toc-linked (depth info &optional scope)
  "Build a table of contents.
     Just like `org-html-toc', except the header is a link to \"#\".
     DEPTH is an integer specifying the depth of the table.  INFO is
     a plist used as a communication channel.  Optional argument SCOPE
     is an element defining the scope of the table.  Return the table
     of contents as a string, or nil if it is empty."
  :override #'org-html-toc
```

```
  (let ((toc-entries
     (mapcar (lambda (headline)
           (cons (org-html--format-toc-headline headline info)
             (org-export-get-relative-level headline info)))
         (org-export-collect-headlines info depth scope))))
    (when toc-entries
      (let ((toc (concat "<div id=\"text-table-of-contents\">"
             (org-html--toc-text toc-entries)
             "</div>\n")))
    (if scope toc
      (let ((outer-tag (if (org-html--html5-fancy-p info)
                   "nav"
                 "div")))
        (concat (format "<%s id=\"table-of-contents\">\n" outer-tag)
            (let ((top-level (plist-get info :html-toplevel-hlevel)))
              (format "<h%d><a href=\"#\" style=\"color:inherit; text-decoration:
              ↪  none;\">%s</a></h%d>\n"
                  top-level
                  (org-html--translate "Table of Contents" info)
                  top-level))
            toc
            (format "</%s>\n" outer-tag))))))))
```

Lastly, let's pile on some metadata. This gives my pages nice embeds.

```
(defun org-html-meta-tags-fancy (info)
  "Use the INFO plist to construct the meta tags, as described in
  ↪  `org-html-meta-tags'."
  (let ((title (org-html-plain-text
                (org-element-interpret-data (plist-get info :title)) info))
        (author (and (plist-get info :with-author)
                     (let ((auth (plist-get info :author)))
                       ;; Return raw Org syntax.
                       (and auth (org-html-plain-text
                                  (org-element-interpret-data auth) info))))))
    (list
     (when (org-string-nw-p author)
       (list "name" "author" author))
     (when (org-string-nw-p (plist-get info :description))
       (list "name" "description"
             (plist-get info :description)))
     '("name" "generator" "org mode")
     '("name" "theme-color" "#77aa99")
     '("property" "og:type" "article")
     (list "property" "og:title" title)
     (let ((subtitle (org-export-data (plist-get info :subtitle) info)))
       (when (org-string-nw-p subtitle)
         (list "property" "og:description" subtitle)))
     '("property" "og:image" "https://tecosaur.com/resources/org/nib.png")
     '("property" "og:image:type" "image/png")
     '("property" "og:image:width" "200")
```

```
     '("property" "og:image:height" "200")
     '("property" "og:image:alt" "Green fountain pen nib")
     (when (org-string-nw-p author)
       (list "property" "og:article:author:first_name" (car (s-split-up-to " "
       ↪  author 2))))
     (when (and (org-string-nw-p author) (s-contains-p " " author))
       (list "property" "og:article:author:last_name" (cadr (s-split-up-to " "
       ↪  author 2))))
     (list "property" "og:article:published_time" (format-time-string "%FT%T%z")))))

(unless (functionp #'org-html-meta-tags-default)
  (defalias 'org-html-meta-tags-default #'ignore))
(setq org-html-meta-tags #'org-html-meta-tags-fancy)
```

Custom CSS/JS The default org HTML export is . . . alright, but we can really jazz it up. lepisma.xyz has a really nice style, and from and org export too! Suffice to say I've snatched it, with a few of my own tweaks applied.

```
<link rel="icon" href="https://tecosaur.com/resources/org/nib.ico" type="image/ico"
↪  />

<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
↪  href="https://tecosaur.com/resources/org/etbookot-roman-webfont.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
↪  href="https://tecosaur.com/resources/org/etbookot-italic-webfont.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
↪  href="https://tecosaur.com/resources/org/Merriweather-TextRegular.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
↪  href="https://tecosaur.com/resources/org/Merriweather-TextItalic.woff2">
<link rel="preload" as="font" crossorigin="anonymous" type="font/woff2"
↪  href="https://tecosaur.com/resources/org/Merriweather-TextBold.woff2">
```

```
(setq org-html-style-plain org-html-style-default
      org-html-htmlize-output-type 'css
      org-html-doctype "html5"
      org-html-html5-fancy t)

(defun org-html-reload-fancy-style ()
  (interactive)
  (setq org-html-style-fancy
        (concat (f-read-text (expand-file-name "misc/org-export-header.html"
        ↪  doom-private-dir))
                "<script>\n"
                (f-read-text (expand-file-name "misc/org-css/main.js"
                ↪  doom-private-dir))
                "</script>\n<style>\n"
                (f-read-text (expand-file-name "misc/org-css/main.css"
                ↪  doom-private-dir))
                "</style>")))
```

```
  (when org-fancy-html-export-mode
    (setq org-html-style-default org-html-style-fancy)))
(org-html-reload-fancy-style)
```

Collapsable src and example blocks By wrapping the `<pre>` element in a `<details>` block, we can obtain collapsable blocks with no CSS, though we will toss a little in anyway to have this looking somewhat spiffy.

Since this collapsability seems useful to have on by default for certain chunks of code, it would be nice if you could set it with `#+attr_html: :collapsed t`.

It would be nice to also have a corresponding global / session-local way of setting this, but I haven't quite been able to get that working (yet).

```
(defvar org-html-export-collapsed nil)
(eval '(cl-pushnew '(:collapsed "COLLAPSED" "collapsed" org-html-export-collapsed t)
                   (org-export-backend-options (org-export-get-backend 'html))))
(add-to-list 'org-default-properties "EXPORT_COLLAPSED")
```

We can take our src block modification a step further, and add a gutter on the side of the src block containing both an anchor referencing the current block, and a button to copy the content of the block.

```
(defadvice! org-html-src-block-collapsable (orig-fn src-block contents info)
  "Wrap the usual <pre> block in a <details>"
  :around #'org-html-src-block
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪  +org-msg-currently-exporting))
      (funcall orig-fn src-block contents info)
    (let* ((properties (cadr src-block))
           (lang (mode-name-to-lang-name
                  (plist-get properties :language)))
           (name (plist-get properties :name))
           (ref (org-export-get-reference src-block info))
           (collapsed-p (member (or (org-export-read-attribute :attr_html src-block
           ↪  :collapsed)
                                    (plist-get info :collapsed))
                                '("y" "yes" "t" t "true" "all"))))
      (format
```

```
       "<details id='%s' class='code'%s><summary%s>%s</summary>
           <div class='gutter'>
           <a href='#%s'>#</a>
           <button title='Copy to clipboard'
           onclick='copyPreToClipbord(this)'>¿</button>\
           </div>
           %s
           </details>"
       ref
       (if collapsed-p "" " open")
       (if name " class='named'" "")
       (if (not name) (concat "<span class='lang'>" lang "</span>")
         (format "<span class='name'>%s</span><span class='lang'>%s</span>" name
         ↪  lang))
       ref
       (if name
           (replace-regexp-in-string (format "<pre\\( class=\"[^\"]+\"\\)?
           ↪  id=\"%s\">" ref) "<pre\\1>"
                                     (funcall orig-fn src-block contents info))
         (funcall orig-fn src-block contents info))))))

(defun mode-name-to-lang-name (mode)
  (or (cadr (assoc mode
                   '(("asymptote" "Asymptote")
                     ("awk" "Awk")
                     ("C" "C")
                     ("clojure" "Clojure")
                     ("css" "CSS")
                     ("D" "D")
                     ("ditaa" "ditaa")
                     ("dot" "Graphviz")
                     ("calc" "Emacs Calc")
                     ("emacs-lisp" "Emacs Lisp")
                     ("fortran" "Fortran")
                     ("gnuplot" "gnuplot")
                     ("haskell" "Haskell")
                     ("hledger" "hledger")
                     ("java" "Java")
                     ("js" "Javascript")
                     ("latex" "LaTeX")
                     ("ledger" "Ledger")
                     ("lisp" "Lisp")
                     ("lilypond" "Lilypond")
                     ("lua" "Lua")
                     ("matlab" "MATLAB")
                     ("mscgen" "Mscgen")
                     ("ocaml" "Objective Caml")
                     ("octave" "Octave")
                     ("org" "Org mode")
                     ("oz" "OZ")
                     ("plantuml" "Plantuml")
                     ("processing" "Processing.js")
                     ("python" "Python")
```

```
                    ("R" "R")
                    ("ruby" "Ruby")
                    ("sass" "Sass")
                    ("scheme" "Scheme")
                    ("screen" "Gnu Screen")
                    ("sed" "Sed")
                    ("sh" "shell")
                    ("sql" "SQL")
                    ("sqlite" "SQLite")
                    ("forth" "Forth")
                    ("io" "IO")
                    ("J" "J")
                    ("makefile" "Makefile")
                    ("maxima" "Maxima")
                    ("perl" "Perl")
                    ("picolisp" "Pico Lisp")
                    ("scala" "Scala")
                    ("shell" "Shell Script")
                    ("ebnf2ps" "ebfn2ps")
                    ("cpp" "C++")
                    ("abc" "ABC")
                    ("coq" "Coq")
                    ("groovy" "Groovy")
                    ("bash" "bash")
                    ("csh" "csh")
                    ("ash" "ash")
                    ("dash" "dash")
                    ("ksh" "ksh")
                    ("mksh" "mksh")
                    ("posh" "posh")
                    ("ada" "Ada")
                    ("asm" "Assembler")
                    ("caml" "Caml")
                    ("delphi" "Delphi")
                    ("html" "HTML")
                    ("idl" "IDL")
                    ("mercury" "Mercury")
                    ("metapost" "MetaPost")
                    ("modula-2" "Modula-2")
                    ("pascal" "Pascal")
                    ("ps" "PostScript")
                    ("prolog" "Prolog")
                    ("simula" "Simula")
                    ("tcl" "tcl")
                    ("tex" "LaTeX")
                    ("plain-tex" "TeX")
                    ("verilog" "Verilog")
                    ("vhdl" "VHDL")
                    ("xml" "XML")
                    ("nxml" "XML")
                    ("conf" "Configuration File"))))
    mode))
```

```
(defun org-html-block-collapsable (orig-fn block contents info)
  "Wrap the usual block in a <details>"
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪  +org-msg-currently-exporting))
      (funcall orig-fn block contents info)
    (let ((ref (org-export-get-reference block info))
          (type (pcase (car block)
                  ('property-drawer "Properties")))
          (collapsed-default (pcase (car block)
                               ('property-drawer t)
                               (_ nil)))
          (collapsed-value (org-export-read-attribute :attr_html block :collapsed))
          (collapsed-p (or (member (org-export-read-attribute :attr_html block
          ↪  :collapsed)
                                   '("y" "yes" "t" t "true"))
                           (member (plist-get info :collapsed) '("all")))))
      (format
       "<details id='%s' class='code'%s>
           <summary%s>%s</summary>
           <div class='gutter'>\
           <a href='#%s'>#</a>
           <button title='Copy to clipboard'
           onclick='copyPreToClipbord(this)'>¿</button>\
           </div>
           %s\n
           </details>"
       ref
       (if (or collapsed-p collapsed-default) "" " open")
       (if type " class='named'" "")
       (if type (format "<span class='type'>%s</span>" type) "")
       ref
       (funcall orig-fn block contents info)))))

(advice-add 'org-html-example-block   :around #'org-html-block-collapsable)
(advice-add 'org-html-fixed-width     :around #'org-html-block-collapsable)
(advice-add 'org-html-property-drawer :around #'org-html-block-collapsable)
```

Include extra font-locking in htmlize Org uses htmlize.el to export buffers with syntax highlighting.

The works fantastically, for the most part. Minor modes that provide font-locking are *not* loaded, and so do not impact the result.

By enabling these modes in `htmlize-before-hook` we can correct this behaviour.

```
(add-hook 'htmlize-before-hook #'highlight-numbers--turn-on)
```

Handle table overflow In order to accommodate wide tables —particularly on mobile devices— we want to set a maximum width and scroll overflow. Unfortunately, this cannot be applied directly to the `table` element, so we have to wrap it in a `div`.

While we're at it, we can a link gutter, as we did with src blocks, and show the `#+name`, if one is given.

```
(defadvice! org-html-table-wrapped (orig-fn table contents info)
  "Wrap the usual <table> in a <div>"
  :around #'org-html-table
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn table contents info)
    (let* ((name (plist-get (cadr table) :name))
           (ref (org-export-get-reference table info)))
      (format "<div id='%s' class='table'>
          <div class='gutter'><a href='#%s'>#</a></div>
          <div class='tabular'>
          %s
          </div>\
          </div>"
              ref ref
              (if name
                  (replace-regexp-in-string (format "<table id=\"%s\"" ref) "<table"
                                            (funcall orig-fn table contents info))
                (funcall orig-fn table contents info))))))
```

TOC as a collapsable tree The TOC is much nicer to navigate as a collapsable tree. Unfortunately we cannot achieve this with CSS alone. Thankfully we can avoid JS though, by adapting the TOC generation code to use a `label` for each item, and a hidden `checkbox` to keep track of state.

To add this, we need to change one line in org-html–format-toc-headline.

Since we can actually accomplish the desired effect by adding advice *around* the function, without overriding it — let's do that to reduce the bug surface of this config a tad.

```
(defadvice! org-html--format-toc-headline-colapseable (orig-fn headline info)
  "Add a label and checkbox to `org-html--format-toc-headline's usual output,
      to allow the TOC to be a collapseable tree."
  :around #'org-html--format-toc-headline
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn headline info)
    (let ((id (or (org-element-property :CUSTOM_ID headline)
                  (org-export-get-reference headline info))))
```

```
      (format "<input type='checkbox' id='toc--%s'/><label for='toc--%s'>%s</label>"
              id id (funcall orig-fn headline info)))))
```

Now, leaves (headings with no children) shouldn't have the `label` item. The obvious way to achieve this is by including some *if no children...* logic in `org-html--format-toc-headline-colapseabl` Unfortunately, I can't my elisp isn't up to par to extract the number of child headings from the mountain of info that org provides.

```
(defadvice! org-html--toc-text-stripped-leaves (orig-fn toc-entries)
  "Remove label"
  :around #'org-html--toc-text
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
  ↪ +org-msg-currently-exporting))
      (funcall orig-fn toc-entries)
    (replace-regexp-in-string "<input [^>]+><label [^>]+>\\(.+?\\)</label></li>"
    ↪ "\\1</li>"
                              (funcall orig-fn toc-entries))))
```

Make verbatim different to code Since we have `verbatim` and `code`, let's make use of the difference.

We can use `code` exclusively for code snippets and commands like: "calling `(message "Hello")` in batch-mode Emacs prints to stdout like `echo`". Then we can use `verbatim` for miscellaneous 'other monospace' like keyboard shortcuts: "either `C-c C-c` or `C-g` is likely the most useful keybinding in Emacs", or file names: "I keep my configuration in `~/.config/doom/`", among other things.

Then, styling these two cases differently can help improve clarity in a document.

```
(setq org-html-text-markup-alist
      '((bold . "<b>%s</b>")
        (code . "<code>%s</code>")
        (italic . "<i>%s</i>")
        (strike-through . "<del>%s</del>")
        (underline . "<span class=\"underline\">%s</span>")
        (verbatim . "<kbd>%s</kbd>")))
```

Change checkbox type We also want to use HTML checkboxes, however we want to get a bit fancier than default

```
(appendq! org-html-checkbox-types
          '((html-span
```

```
            (on . "<span class='checkbox'></span>")
            (off . "<span class='checkbox'></span>")
            (trans . "<span class='checkbox'></span>"))))
(setq org-html-checkbox-type 'html-span)
```

- ☐ I'm yet to do this
- ☐ Work in progress
- ☑ This is done

Header anchors I want to add GitHub-style links on hover for headings.

```
(defun org-export-html-headline-anchor (text backend info)
  (when (and (org-export-derived-backend-p backend 'html)
             org-fancy-html-export-mode)
    (unless (bound-and-true-p +org-msg-currently-exporting)
      (replace-regexp-in-string
       "<h\\([0-9]\\) id=\"\\([a-z0-9-]+\\)\">\\(.*[^ ]\\)<\\/h[0-9]>" ; this is
       ↪ quite restrictive, but due to `org-reference-contraction' I can do this
       "<h\\1 id=\"\\2\">\\3<a aria-hidden=\"true\" href=\"#\\2\">#</a> </h\\1>"
       text))))

(add-to-list 'org-export-filter-headline-functions
             'org-export-html-headline-anchor)
```

It's worth noting that `+org-msg-currently-exporting` is defined in Org Msg.

Acronyms I want to style acronyms nicely. For the sake of convenience in implementation I've actually done this under the LaTeX export section, for the sake of convenance in implementation (this transformation was first added there).

LaTeX Rendering

1. Pre-rendered I consider `dvisvgm` to be a rather compelling option. However this isn't scaled very well at the moment.

   ```
   ;; (setq-default org-html-with-latex `dvisvgm)
   ```

2. MathJax If MathJax is used, we want to use version 3 instead of the default version 2. Looking at a comparison we seem to find that it is ~5 times as fast, uses a single

file instead of multiple, but seems to be a bit bigger unfortunately. Thankfully this can be mitigated my adding the `async` attribute to defer loading.

```
(setq org-html-mathjax-options
      '((path "https://cdn.jsdelivr.net/npm/mathjax@3/es5/tex-svg.js" )
        (scale "1")
        (autonumber "ams")
        (multlinewidth "85%")
        (tagindent ".8em")
        (tagside "right")))

(setq org-html-mathjax-template
      "<script>
          MathJax = {
            chtml: {
              scale: %SCALE
            },
            svg: {
              scale: %SCALE,
              fontCache: \"global\"
            },
            tex: {
              tags: \"%AUTONUMBER\",
              multlineWidth: \"%MULTLINEWIDTH\",
              tagSide: \"%TAGSIDE\",
              tagIndent: \"%TAGINDENT\"
            }
          };
          </script>
          <script id=\"MathJax-script\" async
                  src=\"%PATH\"></script>")
```

Exporting to LaTeX

Compiling By default Org uses `pdflatex` × 3 + `bibtex`. This simply won't do in our modern world. `latexmk` + `biber` (which is used automatically with `latexmk`) is a simply superior combination.

```
;; org-latex-compilers = ("pdflatex" "xelatex" "lualatex"), which are the possible
↪  values for %latex
(setq org-latex-pdf-process '("latexmk -%latex -shell-escape
↪ -interaction=nonstopmode -f -pdf -output-directory=%o %f"))
```

While `org-latex-pdf-process` does support a function, and we could use that instead, this would no longer use the log buffer — it's a bit blind, you give it the file name and expect it to do its thing.

The default values of `org-latex-compilers` is given in commented form to see how `org-latex-pdf-process` works with them.

While the `-%latex` above is slightly hacky (`-pdflatex` expects to be given a value) it allows us to leave `org-latex-compilers` unmodified. This is nice in case I open an org file that uses `#+LATEX_COMPILER` for example, it should still work.

Acronyms I like automatically using spaced small caps for acronyms. For strings I want to be unaffected let's use `;` as a prefix to prevent the transformation — i.e. `;JFK` (as one would want for two-letter geographic locations and names).

While this is the LaTeX section, it's convenient to also provide HTML acronyms here.

```
(defun tec/org-export-latex-filter-acronym (text backend info)
  (let ((the-backend
         (cond
          ((org-export-derived-backend-p backend 'latex) 'latex)
          ((org-export-derived-backend-p backend 'html) 'html)))
        (case-fold-search nil))
    (when the-backend
      (replace-regexp-in-string
       "[;\\\\]?\\b[A-Z][A-Z]+s?"
       (lambda (all-caps-str)
         ;; only format as acronym if str doesn't start with ";" or "\" (for LaTeX
         ↪  commands)
         (cond ((equal (aref all-caps-str 0) ?\;) (substring all-caps-str 1))
               ((equal (aref all-caps-str 0) ?\\) all-caps-str)
               ((equal (aref all-caps-str (- (length all-caps-str) 1)) ?s)
                (pcase the-backend
                  ('latex
                   (concat "\\textls*[70]{\\textsc{" (s-downcase (substring
                   ↪  all-caps-str 0 -1))
                           "}\\protect\\scalebox{.91}[.84]{s}}"))
                  ('html
                   (concat "<span class='acr'>" (substring all-caps-str 0 -1)
                           "</span><small>s</small>"))))
               (t (pcase the-backend
                    ('latex
                     (concat "\\textls*[70]{\\textsc{" (s-downcase all-caps-str)
                     ↪  "}}"))
                    ('html (concat "<span class='acr'>" all-caps-str "</span>")))))))
       text t t))))

(add-to-list 'org-export-filter-plain-text-functions
             'tec/org-export-latex-filter-acronym)
;; FIXME I want to process headings, but this causes issues ATM,
;;       specifically it passes (and formats) the entire section contents
;; (add-to-list 'org-export-filter-headline-functions
;;              'tec/org-export-latex-filter-acronym)
```

Nicer checkboxes

```
(defun +org-export-latex-fancy-item-checkboxes (text backend info)
  (when (org-export-derived-backend-p backend 'latex)
    (replace-regexp-in-string
     "\\\\item\\[{$\\\\|(\\w+\|)$}\\]"
     (lambda (fullmatch)
       (concat "\\\\item[" (pcase (substring fullmatch 9 -3) ; content of capture
       ↪  group
                             ("square"
                             ↪  "\\\\ifdefined\\\\checkboxUnchecked\\\\checkboxUnchecked\\\\else$\\\\sq
                             ↪  )
                             ("boxminus"
                             ↪  "\\\\ifdefined\\\\checkboxTransitive\\\\checkboxTransitive\\\\else$\\\\
                             ("boxtimes"
                             ↪  "\\\\ifdefined\\\\checkboxChecked\\\\checkboxChecked\\\\else$\\\\boxtim
                             ↪  )
                             (_ (substring fullmatch 9 -3))) "]"))
     text)))

(add-to-list 'org-export-filter-item-functions
             '+org-export-latex-fancy-item-checkboxes)
```

Class templates We'll be setting up an nice preamble to use in a new default export class.

```
\\usepackage[T1]{fontenc}\n\
\\usepackage[osf,largesc,helvratio=0.9]{newpxtext}\n\
\\usepackage[scale=0.9]{sourcecodepro}\n\
\\usepackage{bmc-maths}\n\

\\usepackage[activate={true,nocompatibility},final,tracking=true,kerning=true,spacing=true,factor=2000]{
\\usepackage{xcolor}\n\
\\usepackage{booktabs}

\\usepackage{subcaption}
\\usepackage[hypcap=true]{caption}
\\setkomafont{caption}{\\sffamily\\small}
\\setkomafont{captionlabel}{\\upshape\\bfseries}
\\captionsetup{justification=raggedright,singlelinecheck=true}
\\setcapindent{0pt}

\\setlength{\\parskip}{\\baselineskip}\n\
\\setlength{\\parindent}{0pt}\n\

\\AtBeginEnvironment{quote}{\\itshape}

\\usepackage{pifont}
\\newcommand{\\checkboxUnchecked}{$\\square$}
```

```
\\newcommand{\\checkboxTransitive}{\\rlap{\\raisebox{-
↪  0.1ex}{\\hspace{0.35ex}\\Large\\textbf
↪ -}}$\\square$}
\\newcommand{\\checkboxChecked}{\\rlap{\\raisebox{0.2ex}{\\hspace{0.35ex}\\scriptsize
↪  \\ding{52}}}$\\square$}

% args = #1 Name, #2 Colour, #3 Ding, #4 Label
\\newcommand{\\defsimplebox}[4]{%
      \\definecolor{#1}{HTML}{#2}
      \\newenvironment{#1}
      {%
        \\par \\vspace{-0.7\\baselineskip}%
        \\textcolor{#1}{#3} \\textcolor{#1}{\\textbf{#4}}%
        \\vspace{-0.8\\baselineskip}
        \\begin{addmargin}[1em]{1em}
      }{%
        \\end{addmargin}
        \\vspace{-0.5\\baselineskip}
      }%
    }
\\defsimplebox{warning}{e66100}{\\ding{68}}{Warning}
\\defsimplebox{info}{3584e4}{\\ding{68}}{Information}
\\defsimplebox{success}{26a269}{\\ding{68}}{\\vspace{-\\baselineskip}}
\\defsimplebox{error}{c01c28}{\\ding{68}}{Important}
```

The `hyperref` setup needs to be handled separately however.

```
\\colorlet{greenyblue}{blue!70!green}
\\colorlet{blueygreen}{blue!40!green}
\\providecolor{link}{named}{greenyblue}
\\providecolor{cite}{named}{blueygreen}
\\hypersetup{
  pdfauthor={%a},
  pdftitle={%t},
  pdfkeywords={%k},
  pdfsubject={%d},
  pdfcreator={%c},
  pdflang={%L},
  breaklinks=true,
  colorlinks=true,
  linkcolor=,
  urlcolor=link,
  citecolor=cite\n}
\\urlstyle{same}
```

```
(after! ox-latex
  (add-to-list 'org-latex-classes
               '("fancy-article"
```

```
                 "\\documentclass{scrartcl}\n
<<latex-fancy-preamble>>
"
                 ("\\section{%s}" . "\\section*{%s}")
                 ("\\subsection{%s}" . "\\subsection*{%s}")
                 ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
                 ("\\paragraph{%s}" . "\\paragraph*{%s}")
                 ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
  (add-to-list 'org-latex-classes
               '("blank"
                 "[NO-DEFAULT-PACKAGES]
                                    [NO-PACKAGES]
                                    [EXTRA]"
                 ("\\section{%s}" . "\\section*{%s}")
                 ("\\subsection{%s}" . "\\subsection*{%s}")
                 ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
                 ("\\paragraph{%s}" . "\\paragraph*{%s}")
                 ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
  (add-to-list 'org-latex-classes
               '("bmc-article"
                 "\\documentclass[article,code,maths]{bmc}
                                    [NO-DEFAULT-PACKAGES]
                                    [NO-PACKAGES]
                                    [EXTRA]"
                 ("\\section{%s}" . "\\section*{%s}")
                 ("\\subsection{%s}" . "\\subsection*{%s}")
                 ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
                 ("\\paragraph{%s}" . "\\paragraph*{%s}")
                 ("\\subparagraph{%s}" . "\\subparagraph*{%s}")))
  (add-to-list 'org-latex-classes
               '("bmc"
                 "\\documentclass[code,maths]{bmc}
                                    [NO-DEFAULT-PACKAGES]
                                    [NO-PACKAGES]
                                    [EXTRA]"
                 ("\\chapter{%s}" . "\\chapter*{%s}")
                 ("\\section{%s}" . "\\section*{%s}")
                 ("\\subsection{%s}" . "\\subsection*{%s}")
                 ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
                 ("\\paragraph{%s}" . "\\paragraph*{%s}")
                 ("\\subparagraph{%s}" . "\\subparagraph*{%s}"))))

(setq org-latex-default-class "fancy-article"
      org-latex-tables-booktabs t
      org-latex-hyperref-template "
<<latex-fancy-hyperref>>
")
```

A cleverer preamble We always want some particular elements in the preamble, let's call this the "universal preamble"

```
\\usepackage[main,include]{embedall}
\\IfFileExists{./\\jobname.org}{\\embedfile[desc=The original
↪  file]{\\jobname.org}}{}
```

We could have every package we could possibly need in every one of `org-latex-classes`, but that's *horribly* inefficient and I don't want to think about maintaining that.

Instead, we can have a "universal preamble" which contains a snippet which we want to *always* appear, and then conditional preamble snippets, which are only included when a certain regex is successfully found in the Org buffer.

```
(defvar org-latex-universal-preamble "
<<org-latex-universal-preamble>>
"
  "Preamble to be included in every export.")

(defvar org-latex-conditional-preambles
  `((t . org-latex-universal-preamble)
    ("\\[\\[file:.*\\.svg\\]\\]" . "\\usepackage{svg}"))
  "Snippets which are conditionally included in the preamble of a LaTeX export.
      Alist where when the car results in a non-nil value, the cdr is inserted in
      the preamble.  The car may be a:
      - string, which is used as a regex search in the buffer
      - symbol, the value of which used
      - function, the result of the function is used
      The cdr may be a:
      - string, which is inserted without processing
      - symbol, the value of which is inserted
      - function, the result of which is inserted")

(defadvice! org-latex-header-smart-preamble (orig-fn tpl def-pkg pkg snippets-p
↪  &optional extra)
  "Dynamically insert preamble content based on `org-latex-conditional-preambles'."
  :around #'org-splice-latex-header
  (let ((header (funcall orig-fn tpl def-pkg pkg snippets-p extra)))
    (if snippets-p header
      (concat header
              (mapconcat (lambda (term-preamble)
                           (when (pcase (car term-preamble)
                                   ((pred stringp) (save-excursion
                                                     (goto-char (point-min))
                                                     (search-forward-regexp (car
                                                     ↪  term-preamble) nil t)))
                                   ((pred functionp) (funcall (car term-preamble)))
                                   ((pred symbolp) (symbol-value (car
                                   ↪  term-preamble)))
                                   (_ (user-error "org-latex-conditional-preambles
                                   ↪  key %s unable to be used" (car
                                   ↪  term-preamble))))
                             (pcase (cdr term-preamble)
```

```
                                ((pred stringp) (cdr term-preamble))
                                ((pred functionp) (funcall (cdr term-preamble)))
                                ((pred symbolp) (symbol-value (cdr term-preamble)))
                                (_ (user-error "org-latex-conditional-preambles value
                                ↪  %s unable to be used" (cdr term-preamble))))))
                        org-latex-conditional-preambles
                        "\n")))))
```

Pretty code blocks We could just use minted for syntax highlighting — however, we can do better! The `engrave-faces` package lets us use Emacs' font-lock for syntax highlighting, exporting that as LaTeX commands.

```
(setq org-latex-listings 'engraved) ; NOTE non-standard value
```

Thanks to `org-latex-conditional-preambles` and some copy-paste with the `minted` entry in `org-latex-scr-block` we can easily add this as a recognised `org-latex-listings` value.

```
(defadvice! org-latex-src-block-engraved (orig-fn src-block contents info)
  "Like `org-latex-src-block', but supporting an engraved backend"
  :around #'org-latex-src-block
  (if (eq 'engraved (plist-get info :latex-listings))
      (org-latex-scr-block--engraved src-block contents info)
    (funcall orig-fn src-block contents info)))

(defadvice! org-latex-inline-src-block-engraved (orig-fn inline-src-block contents
↪  info)
  "Like `org-latex-inline-src-block', but supporting an engraved backend"
  :around #'org-latex-inline-src-block
  (if (eq 'engraved (plist-get info :latex-listings))
      (org-latex-inline-scr-block--engraved inline-src-block contents info)
    (funcall orig-fn src-block contents info)))

(setq org-latex-engraved-code-preamble "
<<org-latex-engraved-code-preamble>>
")

(add-to-list 'org-latex-conditional-preambles '("#\\+BEGIN_SRC\|\|#\\+begin_src" .
↪  org-latex-engraved-code-preamble) t)
(add-to-list 'org-latex-conditional-preambles '("#\\+BEGIN_SRC\|\|#\\+begin_src" .
↪  engrave-faces-latex-gen-preamble) t)

(defun org-latex-scr-block--engraved (src-block contents info)
  (let* ((lang (org-element-property :language src-block))
         (attributes (org-export-read-attribute :attr_latex src-block))
         (float (plist-get attributes :float))
         (num-start (org-export-get-loc src-block info))
         (retain-labels (org-element-property :retain-labels src-block))
```

```
        (caption (org-element-property :caption src-block))
        (caption-above-p (org-latex--caption-above-p src-block info))
        (caption-str (org-latex--caption/label-string src-block info))
        (placement (or (org-unbracket-string "[" "]" (plist-get attributes
        ↪  :placement))
                       (plist-get info :latex-default-figure-position)))
        (float-env
         (cond
          ((string= "multicolumn" float)
           (format "\\begin{listing*}[%s]\n%s%%s\n%s\\end{listing*}"
                   placement
                   (if caption-above-p caption-str "")
                   (if caption-above-p "" caption-str)))
          (caption
           (format "\\begin{listing}[%s]\n%s%%s\n%s\\end{listing}"
                   placement
                   (if caption-above-p caption-str "")
                   (if caption-above-p "" caption-str)))
          ((string= "t" float)
           (concat (format "\\begin{listing}[%s]\n"
                           placement)
                   "%s\n\\end{listing}"))
          (t "%s")))
        (options (plist-get info :latex-minted-options))
        (content-buffer
         (with-temp-buffer
           (insert
            (let* ((code-info (org-export-unravel-code src-block))
                   (max-width
                    (apply 'max
                           (mapcar 'length
                                   (org-split-string (car code-info)
                                                     "\n")))))
              (org-export-format-code
               (car code-info)
               (lambda (loc _num ref)
                 (concat
                  loc
                  (when ref
                    ;; Ensure references are flushed to the right,
                    ;; separated with 6 spaces from the widest line
                    ;; of code.
                    (concat (make-string (+ (- max-width (length loc)) 6)
                                         ?\s)
                            (format "(%s)" ref)))))
               nil (and retain-labels (cdr code-info))))))
           (funcall (org-src-get-lang-mode lang))
           (engrave-faces-latex-buffer)))
        (content
         (with-current-buffer content-buffer
           (buffer-string)))
        (body
         (format
```

```
           "\\begin{Code}\n\\begin{Verbatim}[%s]\n%s\\end{Verbatim}\n\\end{Code}"
           ;; Options.
           (concat
            (org-latex--make-option-string
             (if (or (not num-start) (assoc "linenos" options))
                 options
               (append
                `(("linenos")
                  ("firstnumber" ,(number-to-string (1+ num-start))))
                options)))
            (let ((local-options (plist-get attributes :options)))
              (and local-options (concat "," local-options))))
           content)))
    (kill-buffer content-buffer)
    ;; Return value.
    (format float-env body)))

(defun org-latex-inline-scr-block--engraved (inline-src-block _contents info)
  (let ((options (org-latex--make-option-string
                  (plist-get info :latex-minted-options)))
        code-buffer code)
    (setq code-buffer
          (with-temp-buffer
            (insert (org-element-property :value inline-src-block))
            (funcall (org-src-get-lang-mode
                      (org-element-property :language inline-src-block)))
            (engrave-faces-latex-buffer)))
    (setq code (with-current-buffer code-buffer
                 (buffer-string)))
    (kill-buffer code-buffer)
    (format "\\Verb%s{%s}"
            (if (string= options "") ""
              (format "[%s]" options))
            code)))
```

Whenever this is used, in order for it to actually work (and look a little better) we add bit to the preamble:

```
\\usepackage{fvextra}
\\fvset{
  commandchars=\\\\\\{\\},
  highlightcolor=white!95!black!80!blue,
  breaklines=true,
  breaksymbol=\\color{white!60!black}\\tiny\\ensuremath{\\hookrightarrow}}
\\renewcommand\\theFancyVerbLine{\\footnotesize\\color{black!40!white}\\arabic{FancyVerbLine}}

% TODO have code boxes keep line vertical alignment
\\usepackage[breakable,xparse]{tcolorbox}
\\DeclareTColorBox[]{Code}{o}%
{colback=white!97!black, colframe=white!94!black,
  fontupper=\\color{EFD}\\footnotesize,
```

```
  IfNoValueTF={#1}%
  {boxsep=2pt, arc=2.5pt, outer arc=2.5pt,
    boxrule=0.5pt, left=2pt}%
  {boxsep=2.5pt, arc=0pt, outer arc=0pt,
    boxrule=0pt, leftrule=1.5pt, left=0.5pt},
  right=2pt, top=1pt, bottom=0.5pt,
  breakable}
```

At some point it would be nice to make the box colours easily customisable. At the moment it's fairly easy to change the syntax highlighting colours with `(setq engrave-faces-preset-styles (engrave-faces-generate-preset))`, but perhaps a toggle which specifies whether to use the default values, the current theme, or any named theme could be a good idea. It should also possible to set the box background dynamically to match. The named theme could work by looking for a style definition with a certain name in a cache dir, and then switching to that theme and producing (and saving) the style definition if it doesn't exist.

Now let's have the example block be styled similarly.

```
(defadvice! org-latex-example-block-engraved (orig-fn example-block contents info)
  "Like `org-latex-example-block', but supporting an engraved backend"
  :around #'org-latex-example-block
  (let ((output-block (funcall orig-fn example-block contents info)))
    (if (eq 'engraved (plist-get info :latex-listings))
        (format "\\begin{Code}[alt]\n%s\n\\end{Code}" output-block)
      output-block)))
```

In addition to the vastly superior visual output, this should also be much faster to for code-heavy documents (like this config).

Performing a little benchmark with this document, I find that this is indeed the case.

LaTeX syntax highlighting backend	Compile time	Overhead	Overhead ratio
verbatim	12 s	0	0.0
lstlistings	15 s	3 s	0.2
Engrave	34 s	22 s	1.8
Pygments (Minted)	184 s	172 s	14.3

Treating the verbatim (no syntax highlighting) result as a baseline; this rudimentary test suggest that `engrave-faces` is around eight times faster than `pygments`, and takes three times as long as no syntax highlighting (verbatim).

Remove non-ascii chars When using `pdflatex`, almost non-ascii characters are generally problematic, and don't appear in the pdf. It's preferable to see that there was *some* character which wasn't displayed as opposed to nothing.

So, as a basic first-pass we replace every non-ascii char with ¿. In future I could add sensible replacements (e.g. turn ¿ into `\S`, and ¿ with `\ldots`).

```
(defun +org-latex-replace-non-ascii-chars (text backend info)
  "Replace non-ascii chars with \\char\"XYZ forms."
  (when (and (org-export-derived-backend-p backend 'latex)
             (string= (plist-get info :latex-compiler) "pdflatex"))
    (replace-regexp-in-string "[^[:ascii:]]" "¿" text)))

(add-to-list 'org-export-filter-final-output-functions
↪  #'+org-latex-replace-non-ascii-chars)
```

Support images from URLs You can link to remote images easily, and they work nicely with HTML-based exports. However, LaTeX can only include local files, and so the current behaviour of `org-latex-link` is just to insert a URL to the image.

We can do better than that by downloading the image to a predictable location, and using that. By making the filename predictable as opposed to just another tempfile, this can provide a caching mechanism.

```
(defadvice! +org-latex-link (orig-fn link desc info)
  "Acts as `org-latex-link', but supports remote images."
  :around #'org-latex-link
  (setq o-link link
        o-desc desc
        o-info info)
  (if (and (member (plist-get (cadr link) :type) '("http" "https"))
           (member (file-name-extension (plist-get (cadr link) :path))
                   '("png" "jpg" "jpeg" "pdf" "svg")))
      (org-latex-link--remote link desc info)
    (funcall orig-fn link desc info)))

(defun org-latex-link--remote (link _desc info)
  (let* ((url (plist-get (cadr link) :raw-link))
         (ext (file-name-extension url))
         (target (format "%s%s.%s"
                         (temporary-file-directory)
                         (replace-regexp-in-string "[./]" "-"
                                                   (file-name-sans-extension
                                                   ↪  (substring (plist-get (cadr
                                                   ↪  link) :path) 2)))
                         ext)))
    (unless (file-exists-p target)
```

```
      (url-copy-file url target))
    (setcdr link (--> (cadr link)
                      (plist-put it :type "file")
                      (plist-put it :path target)
                      (plist-put it :raw-link (concat "file:" target))
                      (list it)))
    (concat "% fetched from " url "\n"
            (org-latex--inline-image link info))))
```

Chameleon — aka. match theme Once the idea of having the look of the LaTeX document produced match the current Emacs theme, I was enraptured. The result is the pseudo-class `chameleon`.

```
(after! ox
  (defvar ox-chameleon-base-class "fancy-article"
    "The base class that chameleon builds on")

  (defvar ox-chameleon--p nil
    "Used to indicate whether the current export is trying to blend in. Set just
    ↪  before being accessed.")

  ;; (setf (alist-get :filter-latex-class
  ;;                  (org-export-backend-filters
  ;;                   (org-export-get-backend 'latex)))
  ;;       'ox-chameleon-latex-class-detector-filter)

  ;; (defun ox-chameleon-latex-class-detector-filter (info backend)
  ;;   ""
  ;;   (setq ox-chameleon--p (when (equal (plist-get info :latex-class)
  ;;                                      "chameleon")
  ;;                           (plist-put info :latex-class ox-chameleon-base-class)
  ;;                           t)))

  ;; TODO make this less hacky. One ideas was as follows
  ;; (map-put (org-export-backend-filters (org-export-get-backend 'latex))
  ;;          :filter-latex-class 'ox-chameleon-latex-class-detector-filter))
  ;; Never seemed to execute though
  (defadvice! ox-chameleon-org-latex-detect (orig-fun info)
    :around #'org-export-install-filters
    (setq ox-chameleon--p (when (equal (plist-get info :latex-class)
                                       "chameleon")
                            (plist-put info :latex-class ox-chameleon-base-class)
                            t))
    (funcall orig-fun info))

  (defadvice! ox-chameleon-org-latex-export (orig-fn info &optional template
  ↪  snippet?)
    :around #'org-latex-make-preamble
    (funcall orig-fn info)
    (if (not ox-chameleon--p)
```

```
      (funcall orig-fn info template snippet?)
    (concat (funcall orig-fn info template snippet?)
            (ox-chameleon-generate-colourings))))

(defun ox-chameleon-generate-colourings ()
  (apply #'format
```

```
          "%% make document follow Emacs theme
              \\definecolor{bg}{HTML}{%s}
              \\definecolor{fg}{HTML}{%s}
              \\definecolor{red}{HTML}{%s}
              \\definecolor{orange}{HTML}{%s}
              \\definecolor{green}{HTML}{%s}
              \\definecolor{teal}{HTML}{%s}
              \\definecolor{yellow}{HTML}{%s}
              \\definecolor{blue}{HTML}{%s}
              \\definecolor{dark-blue}{HTML}{%s}
              \\definecolor{magenta}{HTML}{%s}
              \\definecolor{violet}{HTML}{%s}
              \\definecolor{cyan}{HTML}{%s}
              \\definecolor{dark-cyan}{HTML}{%s}
              \\definecolor{level1}{HTML}{%s}
              \\definecolor{level2}{HTML}{%s}
              \\definecolor{level3}{HTML}{%s}
              \\definecolor{level4}{HTML}{%s}
              \\definecolor{level5}{HTML}{%s}
              \\definecolor{level6}{HTML}{%s}
              \\definecolor{level7}{HTML}{%s}
              \\definecolor{level8}{HTML}{%s}
              \\definecolor{link}{HTML}{%s}
              \\definecolor{cite}{HTML}{%s}
              \\definecolor{itemlabel}{HTML}{%s}
              \\definecolor{code}{HTML}{%s}
              \\definecolor{verbatim}{HTML}{%s}
              \\pagecolor{bg}
              \\color{fg}
              \\addtokomafont{section}{\\color{level1}}
              \\newkomafont{sectionprefix}{\\color{level1}}
              \\addtokomafont{subsection}{\\color{level2}}
              \\newkomafont{subsectionprefix}{\\color{level2}}
              \\addtokomafont{subsubsection}{\\color{level3}}
              \\newkomafont{subsubsectionprefix}{\\color{level3}}
              \\addtokomafont{paragraph}{\\color{level4}}
              \\newkomafont{paragraphprefix}{\\color{level4}}
              \\addtokomafont{subparagraph}{\\color{level5}}
              \\newkomafont{subparagraphprefix}{\\color{level5}}
              \\renewcommand{\\labelitemi}{\\textcolor{itemlabel}{\\textbullet}}
              \\renewcommand{\\labelitemii}{\\textcolor{itemlabel}{\\normalfont\\bfseries
              \\textendash}}
              \\renewcommand{\\labelitemiii}{\\textcolor{itemlabel}{\\textasteriskcentered}}
              \\renewcommand{\\labelitemiv}{\\textcolor{itemlabel}{\\textperiodcentered}}
              \\renewcommand{\\labelenumi}{\\textcolor{itemlabel}{\\theenumi.}}
              \\renewcommand{\\labelenumii}{\\textcolor{itemlabel}{(\\theenumii)}}
              \\renewcommand{\\labelenumiii}{\\textcolor{itemlabel}{\\theenumiii.}}
              \\renewcommand{\\labelenumiv}{\\textcolor{itemlabel}{\\theenumiv.}}
              \\DeclareTextFontCommand{\\texttt}{\\color{code}\\ttfamily}
              \\makeatletter
              \\def\\verbatim@font{\\color{verbatim}\\normalfont\\ttfamily}
              \\makeatother
              %% end customisations
              "
```

```
          (mapcar (doom-rpartial #'substring 1)
                  (list
                   (face-attribute 'solaire-default-face :background)
                   (face-attribute 'default :foreground)
                   ;;
                   (doom-color 'red)
                   (doom-color 'orange)
                   (doom-color 'green)
                   (doom-color 'teal)
                   (doom-color 'yellow)
                   (doom-color 'blue)
                   (doom-color 'dark-blue)
                   (doom-color 'magenta)
                   (doom-color 'violet)
                   (doom-color 'cyan)
                   (doom-color 'dark-cyan)
                   ;;
                   (face-attribute 'outline-1 :foreground)
                   (face-attribute 'outline-2 :foreground)
                   (face-attribute 'outline-3 :foreground)
                   (face-attribute 'outline-4 :foreground)
                   (face-attribute 'outline-5 :foreground)
                   (face-attribute 'outline-6 :foreground)
                   (face-attribute 'outline-7 :foreground)
                   (face-attribute 'outline-8 :foreground)
                   ;;
                   (face-attribute 'link :foreground)
                   (or (face-attribute 'org-ref-cite-face :foreground) (doom-color
                   ↪  'yellow))
                   (face-attribute 'org-list-dt :foreground)
                   (face-attribute 'org-code :foreground)
                   (face-attribute 'org-verbatim :foreground)
                   ))))
 )
```

Make verbatim different to code Since have just gone to so much effort above let's make the most of it by making `verbatim` use `verb` instead of `protectedtexttt` (default).

This gives the same advantages as mentioned in the HTML export section.

```
(setq org-latex-text-markup-alist
      '((bold . "\\textbf{%s}")
        (code . protectedtexttt)
        (italic . "\\emph{%s}")
        (strike-through . "\\sout{%s}")
        (underline . "\\uline{%s}")
        (verbatim . verb)))
```

Exporting to Beamer It's nice to use a different theme

```
(setq org-beamer-theme "[progressbar=foot]metropolis")
```

Then customise it a bit

```

```

And I think that it's natural to divide a presentation into sections, e.g. Introduction, Overview... so let's set bump up the headline level that becomes a frame from `1` to `2`.

```
(setq org-beamer-frame-level 2)
```

Exporting to Markdown When I want to paste exported markdown somewhere (for example when using Emacs Anywhere), it can be preferable to have unicode characters for `---` etc. instead of `&#x2014;`.

To accomplish this, we just need to locally rebind the alist which provides these substitution.

```
(defadvice! org-md-plain-text-unicode-a (orig-fn text info)
  "Locally rebind `org-html-special-string-regexps'"
  :around #'org-md-plain-text
  (let ((org-html-special-string-regexps
         '(("\\\\-" . "-")
           ("---\\([^-]\\)" . "¿\\1")
           ("--\\([^-]\\)" . "¿\\1")
           ("\\.\\.\\." . "¿"))))
    (funcall orig-fn text (plist-put info :with-smart-quotes nil))))
```

In the future, I may want to check `info` to only have this active when `ox-gfm` is being used.

6.3.4 Babel

Doom lazy-loads babel languages, with is lovely.

We need to tell babel to use python3. Who uses python2 anymore anyway? And why doesn't `python` refer to the latest version!?

```
(setq org-babel-python-command "python3")
```

We also like auto-completion here

```
(defun tec-org-python ()
  (if (eq major-mode 'python-mode)
      (progn (anaconda-mode t)
             (company-mode t))))
(add-hook 'org-src-mode-hook 'tec-org-python)
```

6.3.5 ESS

We don't want `R` evaluation to hang the editor, hence

```
(setq ess-eval-visibly 'nowait)
```

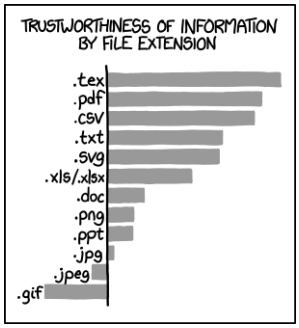
Syntax highlighting is nice, so let's turn all of that on

```
(setq ess-R-font-lock-keywords
      '((ess-R-fl-keyword:keywords . t)
        (ess-R-fl-keyword:constants . t)
        (ess-R-fl-keyword:modifiers . t)
        (ess-R-fl-keyword:fun-defs . t)
        (ess-R-fl-keyword:assign-ops . t)
        (ess-R-fl-keyword:%op% . t)
        (ess-fl-keyword:fun-calls . t)
        (ess-fl-keyword:numbers . t)
        (ess-fl-keyword:operators . t)
        (ess-fl-keyword:delimiters . t)
        (ess-fl-keyword:= . t)
        (ess-R-fl-keyword:F&T . t)))
```

6.4 LaTeX

6.4.1 To-be-implemented ideas

- Paste image from clipboard
  - Determine first folder in `graphicspath` if applicable



File Extensions I have never been lied to by data in a .txt file which has been hand-aligned.

- Ask for file name
- Use `xclip` to save file to graphics folder, or current directory (whichever applies)

```
command -v xclip >/dev/null 2>&1 || { echo >&1 "no xclip"; exit 1; }

if
    xclip -selection clipboard -target image/png -o >/dev/null 2>&1
then
    xclip -selection clipboard -target image/png -o >$1 2>/dev/null
    echo $1
else
    echo "no image"
fi
```

- Insert figure, with filled in details as a result (activate `yasnippet` with filename as variable maybe?)

6.4.2 Compilation

```
(setq TeX-save-query nil
      TeX-show-compilation t
      TeX-command-extra-options "-shell-escape")
(after! latex
  (add-to-list 'TeX-command-list '("XeLaTeX" "%`xelatex%(mode)%' %t" TeX-run-TeX nil
  ↪  t)))
```

For viewing the PDF, I rather like the pdf-tools viewer. While auctex is trying to be nice in recognising that I have some PDF viewing apps installed, I'd rather not have it default to using them, so let's re-order the preferences.

```
(setq +latex-viewers '(pdf-tools evince zathura okular skim sumatrapdf))
```

6.4.3 Snippet helpers

Template For use in the new-file template, let's set out a nice preamble we may want to use. Then let's bind the content to a function, and define some nice helpers.

```
(setq tec/yas-latex-template-preamble "
<<latex-nice-preamble>>
")

(defun tec/yas-latex-get-class-choice ()
  "Prompt user for LaTeX class choice"
  (setq tec/yas-latex-class-choice (ivy-read "Select document class: " '("article"
  ↪  "scrartcl" "bmc") :def "bmc")))

(defun tec/yas-latex-preamble-if ()
  "Based on class choice prompt for insertion of default preamble"
  (if (equal tec/yas-latex-class-choice "bmc") 'nil
    (eq (read-char-choice "Include default preamble? [Type y/n]" '(?y ?n)) ?y)))
```

Deliminators

```
(after! tex
  (defvar tec/tex-last-delim-char nil
    "Last open delim expanded in a tex document")
  (defvar tec/tex-delim-dot-second t
    "When the `tec/tex-last-delim-char' is . a second charachter (this) is prompted
    ↪  for")
  (defun tec/get-open-delim-char ()
    "Exclusivly read next char to tec/tex-last-delim-char"
    (setq tec/tex-delim-dot-second nil)
    (setq tec/tex-last-delim-char (read-char-exclusive "Opening deliminator,
    ↪  recognises: 9 ( [ { < | ."))
    (when (eql ?. tec/tex-last-delim-char)
      (setq tec/tex-delim-dot-second (read-char-exclusive "Other deliminator,
      ↪  recognises: 0 9 (  ) [ ] { } < > |"))))
  (defun tec/tex-open-delim-from-char (&optional open-char)
    "Find the associated opening delim as string"
    (unless open-char (setq open-char (if (eql ?. tec/tex-last-delim-char)
                                          tec/tex-delim-dot-second
                                        tec/tex-last-delim-char)))
    (pcase open-char
      (?\( "(")
      (?9  "(")
      (?\[ "[")
      (?\{ "\\{")
```

```
      (?<  "<")
      (?|  (if tec/tex-delim-dot-second "." "|"))
      (_   ".")))
  (defun tec/tex-close-delim-from-char (&optional open-char)
    "Find the associated closing delim as string"
    (if tec/tex-delim-dot-second
        (pcase tec/tex-delim-dot-second
          (?\) ")")
          (?0  ")")
          (?\] "]")
          (?\} "\\}")
          (?\> ">")
          (?|  "|")
          (_   "."))
      (pcase (or open-char tec/tex-last-delim-char)
        (?\( ")")
        (?9  ")")
        (?\[ "]")
        (?\{ "\\}")
        (?<  ">")
        (?\) ")")
        (?0  ")")
        (?\] "]")
        (?\} "\\}")
        (?\> ">")
        (?|  "|")
        (_   "."))))
  (defun tec/tex-next-char-smart-close-delim (&optional open-char)
    (and (bound-and-true-p smartparens-mode)
         (eql (char-after) (pcase (or open-char tec/tex-last-delim-char)
                             (?\( ?\))
                             (?\[ ?\])
                             (?{ ?})
                             (?< ?>)))))
  (defun tec/tex-delim-yas-expand (&optional open-char)
    (yas-expand-snippet (yas-lookup-snippet "_deliminators" 'latex-mode) (point) (+
    ↪  (point) (if (tec/tex-next-char-smart-close-delim open-char) 2 1))))
```

6.4.4 Editor visuals

Once again, *all hail mixed pitch mode!*

```
(add-hook 'LaTeX-mode-hook #'mixed-pitch-mode)
```

Let's enhance `TeX-fold-math` a bit

```
(after! latex
  (setcar (assoc "¿" LaTeX-fold-math-spec-list) "¿")) ;; make \star bigger

(setq TeX-fold-math-spec-list
      `(;; missing/better symbols
        ("¿" ("le"))
        ("¿" ("ge"))
        ("¿" ("ne"))
        ;; conviniance shorts -- these don't work nicely ATM
        ;; ("¿" ("left"))
        ;; ("¿" ("right"))
        ;; private macros
        ("¿" ("RR"))
        ("¿" ("NN"))
        ("¿" ("ZZ"))
        ("¿" ("QQ"))
        ("¿" ("CC"))
        ("¿" ("PP"))
        ("¿" ("HH"))
        ("¿" ("EE"))
        ("¿" ("dd"))
        ;; known commands
        ("" ("phantom"))
        (,(lambda (num den) (if (and (TeX-string-single-token-p num)
        ↪  (TeX-string-single-token-p den))
                                (concat num "¿" den)
                              (concat "¿" num "¿" den "¿"))) ("frac"))
        (,(lambda (arg) (concat "¿" (TeX-fold-parenthesize-as-neccesary arg)))
        ↪  ("sqrt"))
        (,(lambda (arg) (concat "¿" (TeX-fold-parenthesize-as-neccesary arg)))
        ↪  ("vec"))
        ("¿*{1}*¿" ("text"))
        ;; private commands
        ("|{1}|" ("abs"))
        ("¿{1}¿" ("norm"))
        ("¿{1}¿" ("floor"))
        ("¿{1}¿" ("ceil"))
        ("¿{1}¿" ("round"))
        ("¿{1}/¿{2}" ("dv"))
        ("¿{1}/¿{2}" ("pdv"))
        ;; fancification
        ("{1}" ("mathrm"))
        (,(lambda (word) (string-offset-roman-chars 119743 word)) ("mathbf"))
        (,(lambda (word) (string-offset-roman-chars 119951 word)) ("mathcal"))
        (,(lambda (word) (string-offset-roman-chars 120003 word)) ("mathfrak"))
        (,(lambda (word) (string-offset-roman-chars 120055 word)) ("mathbb"))
        (,(lambda (word) (string-offset-roman-chars 120159 word)) ("mathsf"))
        (,(lambda (word) (string-offset-roman-chars 120367 word)) ("mathtt"))
        )
      TeX-fold-macro-spec-list
      '(
        ;; as the defaults
        ("[f]" ("footnote" "marginpar"))
```

```
        ("[c]" ("cite"))
        ("[l]" ("label"))
        ("[r]" ("ref" "pageref" "eqref"))
        ("[i]" ("index" "glossary"))
        ("..." ("dots"))
        ("{1}" ("emph" "textit" "textsl" "textmd" "textrm" "textsf" "texttt"
                "textbf" "textsc" "textup"))
        ;; tweaked defaults
        ("¿" ("copyright"))
        ("¿" ("textregistered"))
        ("¿"  ("texttrademark"))
        ("[1]:||¿" ("item"))
        ("¿¿¿{1}" ("part" "part*"))
        ("¿¿{1}" ("chapter" "chapter*"))
        ("¿¿{1}" ("section" "section*"))
        ("¿¿¿{1}" ("subsection" "subsection*"))
        ("¿¿¿¿{1}" ("subsubsection" "subsubsection*"))
        ("¿¿{1}" ("paragraph" "paragraph*"))
        ("¿¿¿{1}" ("subparagraph" "subparagraph*"))
        ;; extra
        ("¿¿{1}" ("begin"))
        ("¿¿{1}" ("end"))
        ))

(defun string-offset-roman-chars (offset word)
  "Shift the codepoint of each charachter in WORD by OFFSET with an extra -6 shift
  ↪  if the letter is lowercase"
  (apply 'string
         (mapcar (lambda (c)
                   (string-offset-apply-roman-char-exceptions
                    (+ (if (>= c 97) (- c 6) c) offset)))
                 word)))

(defvar string-offset-roman-char-exceptions
  '(;; lowercase serif
    (119892 .  8462) ; ¿
    ;; lowercase caligraphic
    (119994 . 8495) ; ¿
    (119996 . 8458) ; ¿
    (120004 . 8500) ; ¿
    ;; caligraphic
    (119965 . 8492) ; ¿
    (119968 . 8496) ; ¿
    (119969 . 8497) ; ¿
    (119971 . 8459) ; ¿
    (119972 . 8464) ; ¿
    (119975 . 8466) ; ¿
    (119976 . 8499) ; ¿
    (119981 . 8475) ; ¿
    ;; fraktur
    (120070 . 8493) ; ¿
    (120075 . 8460) ; ¿
    (120076 . 8465) ; ¿
```

```
    (120085 . 8476) ; ¿
    (120092 . 8488) ; ¿
    ;; blackboard
    (120122 . 8450) ; ¿
    (120127 . 8461) ; ¿
    (120133 . 8469) ; ¿
    (120135 . 8473) ; ¿
    (120136 . 8474) ; ¿
    (120137 . 8477) ; ¿
    (120145 . 8484) ; ¿
    )
  "An alist of deceptive codepoints, and then where the glyph actually resides.")

(defun string-offset-apply-roman-char-exceptions (char)
  "Sometimes the codepoint doesn't contain the char you expect.
      Such special cases should be remapped to another value, as given in
  ↪   `string-offset-roman-char-exceptions'."
  (if (assoc char string-offset-roman-char-exceptions)
      (cdr (assoc char string-offset-roman-char-exceptions))
    char))

(defun TeX-fold-parenthesize-as-neccesary (tokens &optional suppress-left
↪  suppress-right)
  "Add ¿ ¿ parenthesis as if multiple LaTeX tokens appear to be present"
  (if (TeX-string-single-token-p tokens) tokens
    (concat (if suppress-left "" "¿")
            tokens
            (if suppress-right "" "¿"))))

(defun TeX-string-single-token-p (teststring)
  "Return t if TESTSTRING appears to be a single token, nil otherwise"
  (if (string-match-p "^\\\\?\\w+$" teststring) t nil))
```

Some local keybindings to make life a bit easier

```
(after! tex
  (map!
   :map LaTeX-mode-map
   :ei [C-return] #'LaTeX-insert-item)
  (setq TeX-electric-math '("\\(" . "")))
```

Maths deliminators can be de-emphasised a bit

```
;; Making \( \) less visible
(defface unimportant-latex-face
  '((t
     :inherit font-lock-comment-face :family "Overpass" :weight light))
  "Face used to make \\(\\), \\[\\] less visible."
  :group 'LaTeX-math)
```

```
(font-lock-add-keywords
 'latex-mode
 `((,(rx (and "\\" (any "()[]"))) 0 'unimportant-latex-face prepend))
 'end)

(font-lock-add-keywords
 'latex-mode
 `((,"\\\\[[:word:]]+" 0 'font-lock-keyword-face prepend))
 'end)
```

And enable shell escape for the preview

```
(setq preview-LaTeX-command '("%`%l \"\\nonstopmode\\nofiles\
    \\PassOptionsToPackage{" "," . preview-required-option-list) "}{preview}\
    \\AtBeginDocument{\\ifx\\ifPreview\\undefined"
preview-default-preamble "\\fi}\"%' \"\\detokenize{\" %t \"}\""))
```

6.4.5 CDLaTeX

The symbols and modifies are very nice by default, but could do with a bit of fleshing out. Let's change the prefix to a key which is similarly rarely used, but more convenient, like `;`.

```
(after! cdlatex
  (setq ;; cdlatex-math-symbol-prefix ?\; ;; doesn't work at the moment :(
   cdlatex-math-symbol-alist
   '( ;; adding missing functions to 3rd level symbols
     (?_    ("\\downarrow"  ""           "\\inf"))
     (?2    ("^2"           "\\sqrt{?}"    ""     ))
     (?3    ("^3"           "\\sqrt[3]{?}"  ""     ))
     (?^    ("\\uparrow"    ""           "\\sup"))
     (?k    ("\\kappa"      ""           "\\ker"))
     (?m    ("\\mu"         ""           "\\lim"))
     (?c    (""             "\\circ"     "\\cos"))
     (?d    ("\\delta"      "\\partial"  "\\dim"))
     (?D    ("\\Delta"      "\\nabla"    "\\deg"))
     ;; no idea why \Phi isnt on 'F' in first place, \phi is on 'f'.
     (?F    ("\\Phi"))
     ;; now just conveniance
     (?.    ("\\cdot" "\\dots"))
     (?:    ("\\vdots" "\\ddots"))
     (?*    ("\\times" "\\star" "\\ast")))
   cdlatex-math-modify-alist
   '( ;; my own stuff
     (?B    "\\mathbb"        nil          t    nil  nil)
     (?a    "\\abs"           nil          t    nil  nil))))
```

6.4.6 SyncTeX

```
(after! tex
  (add-to-list 'TeX-view-program-list '("Evince" "evince %o"))
  (add-to-list 'TeX-view-program-selection '(output-pdf "Evince")))
```

6.4.7 Fixes

In case of Emacs28,

```
(when EMACS28+
  (add-hook 'latex-mode-hook #'TeX-latex-mode))
```

6.5 Python

Since I'm using `mypyls`, as suggested in :lang python LSP support I'll tweak the priority of `mypyls`

```
(after! lsp-python-ms
  (set-lsp-priority! 'mspyls 1))
```

6.6 R

6.6.1 Editor Visuals

```
(after! ess-r-mode
  (appendq! +ligatures-extra-symbols
            '(:assign "¿"
              :multiply "¿"))
  (set-ligatures! 'ess-r-mode
    ;; Functional
    :def "function"
    ;; Types
    :null "NULL"
    :true "TRUE"
    :false "FALSE"
    :int "int"
    :floar "float"
    :bool "bool"
```

```
    ;; Flow
    :not "!"
    :and "&&" :or "||"
    :for "for"
    :in "%in%"
    :return "return"
    ;; Other
    :assign "<-"
    :multiply "%*%"))
```

6.7 Graphviz

```
(use-package! graphviz-dot-mode
  :commands graphviz-dot-mode
  :mode ("\\.dot\\'" "\\.gz\\'")
  :init
  (after! org
    (setcdr (assoc "dot" org-src-lang-modes)
            'graphviz-dot)))

(use-package! company-graphviz-dot
  :after graphviz-dot-mode)
```

6.8 Markdown

Let's use mixed pitch, because it's great

```
(add-hook! (gfm-mode markdown-mode) #'mixed-pitch-mode)
```

Most of the time when I write markdown, it's going into some app/website which will do it's own line wrapping, hence we *only* want to use visual line wrapping. No hard stuff.

```
(add-hook! (gfm-mode markdown-mode) #'visual-line-mode #'turn-off-auto-fill)
```

Since markdown is often seen as rendered HTML, let's try to somewhat mirror the style or markdown renderers.

Most markdown renders seem to make the first three headings levels larger than normal text, the first two much so. Then the fourth level tends to be the same as body text, while the fifth and sixth are (increasingly) smaller, with the sixth greyed out. Since the sixth level is so small, I'll turn up the boldness a notch.

```
(custom-set-faces!
  '(markdown-header-face-1 :height 1.25 :weight extra-bold :inherit
  ↪  markdown-header-face)
  '(markdown-header-face-2 :height 1.15 :weight bold       :inherit
  ↪  markdown-header-face)
  '(markdown-header-face-3 :height 1.08 :weight bold       :inherit
  ↪  markdown-header-face)
  '(markdown-header-face-4 :height 1.00 :weight bold       :inherit
  ↪  markdown-header-face)
  '(markdown-header-face-5 :height 0.90 :weight bold       :inherit
  ↪  markdown-header-face)
  '(markdown-header-face-6 :height 0.75 :weight extra-bold :inherit
  ↪  markdown-header-face))
```

6.9 Beancount

There are a number of rather compelling advantages to plain text accounting, with ledger being the most obvious example. However, beancount, a more recent implementation of the idea is ledger-compatable (meaning I can switch easily if I change my mind) and has a gorgeous front-end — fava.

Of course, there's an Emacs mode for this.

```
(use-package! beancount
  :mode ("\\.beancount\\'" . beancount-mode)
  :init
  (after! all-the-icons
    (add-to-list 'all-the-icons-icon-alist
                 '("\\.beancount\\'" all-the-icons-material "attach_money" :face
                 ↪  all-the-icons-lblue))
    (add-to-list 'all-the-icons-mode-icon-alist
                 '(beancount-mode all-the-icons-material "attach_money" :face
                 ↪  all-the-icons-lblue)))
  :config
  (setq beancount-electric-currency t)
  (defun beancount-bal ()
    "Run bean-report bal."
    (interactive)
    (let ((compilation-read-command nil))
      (beancount--run "bean-report"
                      (file-relative-name buffer-file-name) "bal")))
  (map! :map beancount-mode-map
        :n "TAB" #'beancount-align-to-previous-number
        :i "RET" (cmd! (newline-and-indent) (beancount-align-to-previous-number))))
```

6.10 Snippets

6.10.1 latex mode

File Template

```
# -*- mode: snippet -*-
# name: LaTeX template
↪
# --
\documentclass${1:[${2:opt1,...}]}{`(tec/yas-latex-get-class-choice)`}

\title{${3:`(s-titleized-words (file-name-base buffer-file-name))`}}
\author{${4:`(user-full-name)`}}
\date{${5:`(format-time-string "%Y-%m-%d")`}}
`(if (tec/yas-latex-preamble-if) tec/yas-latex-template-preamble "")`
\begin{document}

\maketitle

$0

\end{document}
```

deliminators

```
# name: _deliminators
↪
# --
\left`(tec/tex-open-delim-from-char)` `%`$1 \right`(tec/tex-close-delim-from-char)`
↪  $0
```

aligned equals

```
# key: ==
# name: aligned equals
# --
&=
```

begin alias

```
# -*- mode: snippet -*-
# name: begin-alias
# key: beg
```

```
# type: command
# --
(doom-snippets-expand :name "begin")
```

cases

```
# -*- mode: snippet -*-
# key: cs
# name: cases
# group: math
# condition: (texmathp)
# --
\begin{cases}
  `%`$1
\end{cases}$0
```

code

```
# -*- mode: snippet -*-
# name: code
# --
\begin{minted}{${1:language}}
${0:`%`}
\end{minted}
```

corollary

```
# -*- mode: snippet -*-
# name: corollary
# key: clr
# group: theorems
# --
\begin{corollary}${1:[${2:name}]}
  `%`$0
\end{corollary}
```

definition

```
# -*- mode: snippet -*-
# name: definition
# key: def
# group: theorems
# --
\begin{definition}${1:[${2:name}]}
  `%`$0
\end{definition}
```

deliminators

```
# -*- mode: snippet -*-
# name: deliminators
# key: @
# condition: (texmathp)
# type: command
# --
(tec/get-open-delim-char)
(yas-expand-snippet (yas-lookup-snippet "_deliminators" 'latex-mode))
```

deliminators angle

```
# -*- mode: snippet -*-
# name: deliminators - angle <>
# key: <
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\<)
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

deliminators bracket

```
# -*- mode: snippet -*-
# name: deliminators - bracket []
# key: [
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\[)
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

deliminators curly

```
# -*- mode: snippet -*-
# name: deliminators - curley {}
# key: {
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\{)
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

deliminators paren

```
# -*- mode: snippet -*-
# name: deliminators - paren ()
# key: (
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\()
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

enumerate

```
# -*- mode: snippet -*-
# name: enumerate
# key: en
# --
\begin{enumerate}
`(if % % "  \\item ")`$0
\end{enumerate}
```

example

```
# -*- mode: snippet -*-
# name: example
# key: eg
# group: theorems
# --
\begin{example}${1:[${2:name}]}
  `%`$0
\end{example}
```

frac short

```
# -*- mode: snippet -*-
# key: /
# name: frac-short
# group: math
# condition: (texmathp)
# --
\frac{${1:`(or % "")`}}{$2}$0
```

int ^

```
# -*- mode: snippet -*-
# key: int
# name: int_^
# --
\int${1:$(when (> (length yas-text) 0) "_")
}${1:$(when (> (length yas-text) 1) "{")
}${1:left}${1:$(when (> (length yas-text) 1) "}")
}${2:$(when (> (length yas-text) 0) "^")
}${2:$(when (> (length yas-text) 1) "{")
}${2:right}${2:$(when (> (length yas-text) 1) "}")} $0
```

itemize

```
# -*- mode: snippet -*-
# name: itemize
# key: it
# uuid: it
# --
\begin{itemize}
`(if % % "  \\item ")`$0
\end{itemize}
```

lemma

```
# -*- mode: snippet -*-
# name: lemma
# key: lmm
# group: theorems
# --
\begin{lemma}${1:[${2:name}]}
  `%`$0
\end{lemma}
```

lim

```
# -*- mode: snippet -*-
# name: lim
# key: lim
# --
\lim_{${1:n} \to ${2:\infty}} $0
```

mathclap

```
# -*- mode: snippet -*-
# key: mc
# name: mathclap
# group: math
```

```
# condition: (texmathp)
# --
\mathclap{`%`$1}$0
```

prod ^

```
# key: prod
# name: prod_^
# --
\prod${1:$(when (> (length yas-text) 0) "_")
}${1:$(when (> (length yas-text) 1) "{")
}${1:i=1}${1:$(when (> (length yas-text) 1) "}")
}${2:$(when (> (length yas-text) 0) "^")
}${2:$(when (> (length yas-text) 1) "{")
}${2:n}${2:$(when (> (length yas-text) 1) "}")} $0
```

proof

```
# -*- mode: snippet -*-
# name: proof
# key: prf
# group: theorems
# --
\begin{proof}${1:[${2:name}]}
  `%`$0
\end{proof}
```

remark

```
# -*- mode: snippet -*-
# name: remark
# key: rmk
# group: theorems
# --
\begin{remark}${1:[${2:name}]}
  `%`$0
\end{remark}
```

sum ^

```
# key: sum
# name: sum_^
# --
\sum${1:$(when (> (length yas-text) 0) "_")
}${1:$(when (> (length yas-text) 1) "{")
}${1:i=1}${1:$(when (> (length yas-text) 1) "}")
}${2:$(when (> (length yas-text) 0) "^")
```

```
}${2:$(when (> (length yas-text) 1) "{")
}${2:n}${2:$(when (> (length yas-text) 1) "}")} $0
```

theorem

```
# -*- mode: snippet -*-
# name: theorem
# key: thm
# group: theorems
# --
\begin{theorem}${1:[${2:name}]}
  `%`$0
\end{theorem}
```

6.10.2 markdown mode

File Template

```
# -*- mode: snippet -*-
# name: Org template
# --
# ${1:`(s-titleized-words (file-name-base buffer-file-name))`}

$0
```

6.10.3 org mode

File Template

```
# -*- mode: snippet -*-
# name: Org template
# --
#+title: ${1:`(s-titleized-words (file-name-base buffer-file-name))`}
#+author: ${2:`(user-full-name)`}
#+date: ${3: `(format-time-string "%Y-%m-%d")`}

$0
```

display maths

```
# -*- mode: snippet -*-
# name: display math
# key: M
# condition: t
# expand-env: ((yas-after-exit-snippet-hook (lambda () (org-edit-special)
↪  (evil-insert-state) (insert "\n  \n") (left-char))))
# --
\\[`%`$0\\]
```

elisp src

```
# -*- mode: snippet -*-
# name: elisp src
# uuid: src_elisp
# key: <el
# condition: t
# expand-env: ((yas-after-exit-snippet-hook #'org-edit-src-code))
# --
#+begin_src emacs-lisp
`%`$0
#+end_src
```

global property

```
# -*- mode: snippet -*-
# name: Global property
# key: #+p
# condition: (> 20 (line-number-at-pos))
# --
#+property: $0
```

header arg dir

```
# -*- mode: snippet -*-
# name: Header arg - dir
# key: d
# condition: (+yas/org-src-header-p)
# --
:dir `(file-relative-name (read-directory-name "Working directory: "))` $0
```

header arg eval

```
# -*- mode: snippet -*-
# name: Header arg - eval
# key: v
# condition: (+yas/org-src-header-p)
```

```
# --
:eval `(ivy-read "Evaluate: " '("no" "query" "no-export" "query-export"))` $0
```

header arg export

```
# -*- mode: snippet -*-
# name: Header arg - export
# key: e
# condition: (+yas/org-src-header-p)
# --
:exports `(ivy-read "Exports: " '("code" "results" "both" "none"))` $0
```

header arg results

```
# -*- mode: snippet -*-
# name: Header arg - results
# key: r
# condition: (+yas/org-src-header-p)
# --
:results `
(replace-regexp-in-string "(default)" ""
(concat
    (ivy-read "Result collection: " '("(default)" "value " "output ") :preselect
    ↪  "(default)")
    (ivy-read "Result type: " '("(default)" "table " "vector " "list " "verbatim "
    ↪  "file ") :preselect "(default)")
    (ivy-read "Result format: " '("(default)" "code " "drawer " "html " "latex "
    ↪  "link " "graphics " "org " "pp " "raw ") :preselect "(default)")
    (ivy-read "Result handling: " '("(default)" "silent " "replace " "append "
    ↪  "prepend ") :preselect "(default)")))
`$0
```

header arg session

```
# -*- mode: snippet -*-
# name: Header arg - session
# key: s
# condition: (+yas/org-src-header-p)
# --
:session "${1:`(file-name-base (buffer-file-name))`-session}" $0
```

inline math

```
# -*- mode: snippet -*-
# name: inline math
# key: m
```

```
# condition: t
# expand-env: ((yas-after-exit-snippet-hook (lambda () (org-edit-special)
↪  (evil-insert-state))))
# --
\\(`%`$0\\)
```

property header args

```
# -*- mode: snippet -*-
# name: Property - header arg
# key: h
# condition: (or (looking-back "^#\\+PROPERTY:.*" (line-beginning-position))
↪  (looking-back "^#\\+property:.*" (line-beginning-position)))
# --
header-args:${1:`(or (+yas/org-most-common-no-property-lang) "?")`} $0
```

python src

```
# -*- mode: snippet -*-
# name: python src
# uuid: src_python
# key: <py
# condition: t
# expand-env: ((yas-after-exit-snippet-hook #'org-edit-src-code))
# --
#+begin_src python
`%`$0
#+end_src
```

src

```
# -*- mode: snippet -*-
# name: #+begin_src
# uuid: src
# key: src
# --
#+begin_src ${1:`(+yas/org-last-src-lang)`}
`%`$0
#+end_src
```