

# Doom Emacs Configuration

*The Methods, Management, and Menagerie  
of Madness — in meticulous detail*

TECOSAUR

d1e3042  
2021-11-20  
17:44 UTC



# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Why Emacs? . . . . .	7
1.1.1	The enveloping editor . . . . .	8
1.1.2	Some notably unique features . . . . .	8
1.1.3	Issues . . . . .	9
1.1.4	Teach a man to fish. . . . .	9
1.2	Editor comparison . . . . .	10
1.3	Notes for the unwary adventurer . . . . .	12
1.3.1	Extra Requirements . . . . .	12
1.4	Current Issues . . . . .	13
1.4.1	Magit push in daemon . . . . .	13
1.4.2	Unread emails doesn't work across Emacs instances . . . . .	14
<b>2</b>	<b>Rudimentary configuration</b>	<b>15</b>
2.1	Personal Information . . . . .	15
2.2	Better defaults . . . . .	15
2.2.1	Simple settings . . . . .	15
2.2.2	Frame sizing . . . . .	16
2.2.3	Auto-customisations . . . . .	16
2.2.4	Windows . . . . .	17
2.2.5	Buffer defaults . . . . .	17
2.3	Doom configuration . . . . .	18
2.3.1	Modules . . . . .	18
2.3.2	Visual Settings . . . . .	23
2.3.3	Some helper macros . . . . .	26
2.3.4	Allow babel execution in CLI actions . . . . .	26
2.3.5	Elisp REPL . . . . .	27
2.3.6	Asynchronous config tangling . . . . .	29
2.3.7	Dashboard quick actions . . . . .	30
2.4	Other things . . . . .	31
2.4.1	Editor interaction . . . . .	31
2.4.2	Window title . . . . .	31
2.4.3	Splash screen . . . . .	32
2.4.4	Systemd daemon . . . . .	38
2.4.5	Emacs client wrapper . . . . .	39
2.4.6	Prompt to run setup script . . . . .	41

<b>3</b>	<b>Packages</b>	<b>43</b>
3.1	Loading instructions	43
3.1.1	Packages in MELPA/ELPA/emacsmirror	43
3.1.2	Packages from git repositories	43
3.1.3	Disabling built-in packages	44
3.2	Convenience	44
3.2.1	Avy	44
3.2.2	Rotate (window management)	44
3.2.3	Emacs Everywhere	45
3.2.4	Which-key	45
3.3	Tools	46
3.3.1	Abbrev	46
3.3.2	Very large files	46
3.3.3	Eros	47
3.3.4	EVIL	47
3.3.5	Consult	47
3.3.6	Magit	48
3.3.7	Magit delta	50
3.3.8	Smerge	51
3.3.9	Company	52
3.3.10	Projectile	53
3.3.11	Ispell	53
3.3.12	TRAMP	54
3.3.13	Auto activating snippets	55
3.3.14	Screenshot	56
3.3.15	Etrace	56
3.3.16	YASnippet	56
3.3.17	String inflection	57
3.3.18	Smart parentheses	57
3.4	Visuals	58
3.4.1	Info colours	58
3.4.2	Modus themes	59
3.4.3	Theme magic	59
3.4.4	Emojify	61
3.4.5	Doom modeline	62
3.4.6	Keycast	63
3.4.7	Screencast	64
3.4.8	Mixed pitch	65
3.4.9	Marginalia	66
3.4.10	Centaur Tabs	67

3.4.11	All the icons . . . . .	68
3.4.12	Prettier page breaks . . . . .	68
3.4.13	Writeroom . . . . .	68
3.4.14	Treemacs . . . . .	70
3.5	Frivolities . . . . .	72
3.5.1	xkcd . . . . .	72
3.5.2	Selectric . . . . .	78
3.5.3	Wttrin . . . . .	78
3.5.4	Spray . . . . .	79
3.5.5	Elcord . . . . .	79
3.6	File types . . . . .	80
3.6.1	Authinfo . . . . .	80
3.6.2	Systemd . . . . .	80
3.6.3	Stan . . . . .	80
<b>4</b>	<b>Applications</b>	<b>81</b>
4.1	Ebooks . . . . .	81
4.2	Calculator . . . . .	85
4.2.1	Defaults . . . . .	85
4.2.2	CalcTeX . . . . .	85
4.2.3	Embedded calc . . . . .	86
4.3	IRC . . . . .	88
4.3.1	Org-style emphasis . . . . .	90
4.3.2	Emojis . . . . .	90
4.4	Newsfeed . . . . .	94
4.4.1	Keybindings . . . . .	95
4.4.2	Usability enhancements . . . . .	95
4.4.3	Visual enhancements . . . . .	96
4.4.4	Functionality enhancements . . . . .	98
4.5	Dictionary . . . . .	99
4.6	Mail . . . . .	101
4.6.1	Fetching . . . . .	101
4.6.2	Indexing/Searching . . . . .	113
4.6.3	Sending . . . . .	114
4.6.4	Mu4e . . . . .	116
4.6.5	Org Msg . . . . .	123
<b>5</b>	<b>Language configuration</b>	<b>124</b>
5.1	General . . . . .	124
5.1.1	File Templates . . . . .	124

5.2	Plaintext . . . . .	124
5.3	Org . . . . .	124
5.3.1	System config . . . . .	126
5.3.2	Packages . . . . .	127
5.3.3	Behaviour . . . . .	132
5.3.4	Visuals . . . . .	164
5.3.5	Exporting . . . . .	178
5.3.6	HTML Export . . . . .	180
5.3.7	L <sup>A</sup> T <sub>E</sub> X Export . . . . .	197
5.3.8	Beamer Export . . . . .	234
5.3.9	Reveal export . . . . .	234
5.3.10	ASCII export . . . . .	234
5.3.11	Markdown Export . . . . .	236
5.3.12	Babel . . . . .	237
5.3.13	ESS . . . . .	238
5.4	L <sup>A</sup> T <sub>E</sub> X . . . . .	239
5.4.1	To-be-implemented ideas . . . . .	239
5.4.2	Compilation . . . . .	240
5.4.3	Snippet helpers . . . . .	240
5.4.4	Editor visuals . . . . .	242
5.4.5	Math input . . . . .	246
5.4.6	SyncTeX . . . . .	247
5.4.7	Fixes . . . . .	247
5.5	Python . . . . .	247
5.6	PDF . . . . .	248
5.6.1	MuPDF . . . . .	248
5.6.2	Terminal viewing . . . . .	248
5.7	R . . . . .	249
5.7.1	Editor Visuals . . . . .	249
5.8	Julia . . . . .	250
5.9	Graphviz . . . . .	250
5.10	Markdown . . . . .	250
5.11	Beancount . . . . .	251
5.12	Snippets . . . . .	252
5.12.1	Latex mode . . . . .	252
5.12.2	Markdown mode . . . . .	258
5.12.3	Org mode . . . . .	258

---

*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*

# CHAPTER 1 Introduction

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

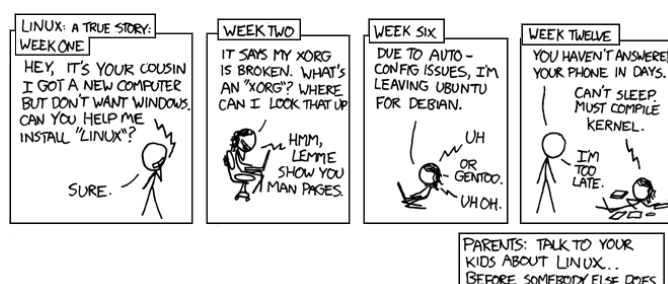
*Reddit1: I just discovered this thing, isn't it cool.*

*Reddit2: 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

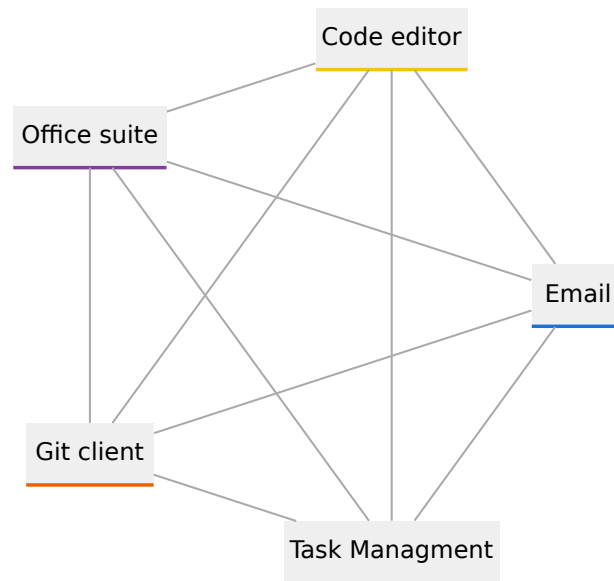
Emacs can be thought of as a platform within which various elements of your workflow may settle, with the potential for rich integrations between them — a *life IDE* if you will.

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.

### 1.1.2 Some notably unique features

- Recursive editing
- Completely introspectable, with pervasive docstrings





**Figure 1.1:** Some sample workflow integrations that can be used within Emacs

- 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))
```

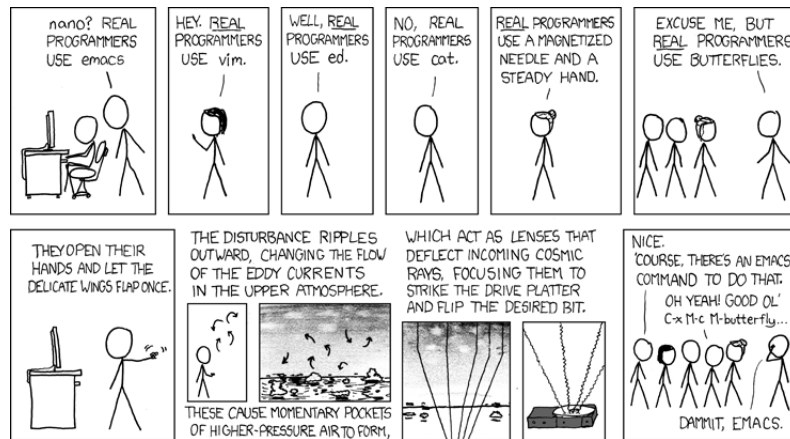
Want to test it out? You don't need to save and restart, you can just *evaluate the expression* within your current Emacs instance and try it immediately! This editor is, after all, a Lisp interpreter.

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

## 1.2 Editor comparison

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

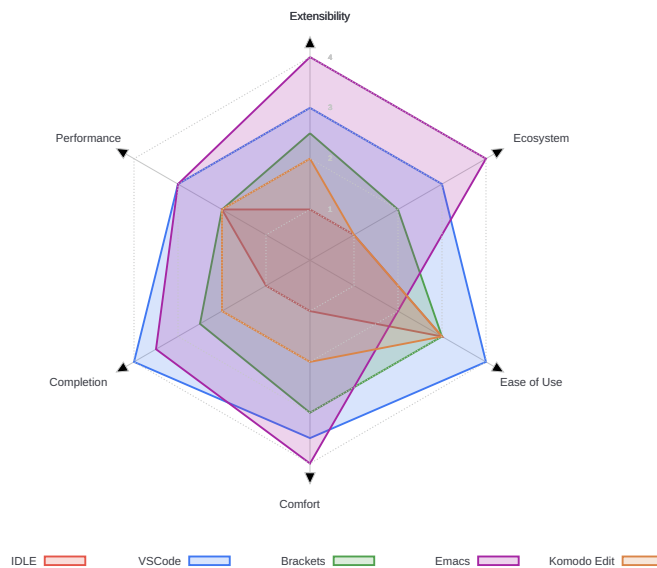


**Real Programmers** Real programmers set the universal constants at the start such that the universe evolves to contain the disk with the data they want.

- 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.

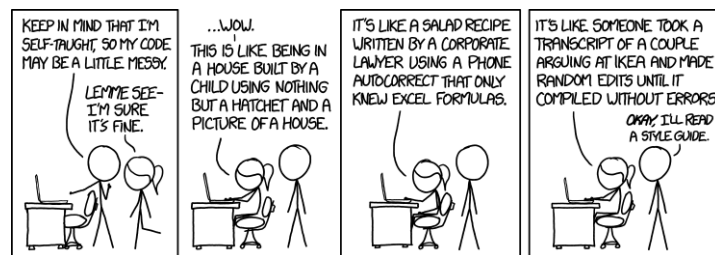
If you are so bold as to wish to steal bits of my config (or if I upgrade and wonder why things aren't working), here's a list of sections which rely on external setup (i.e. outside of this config).

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

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 :)

### 1.3.1 Extra Requirements

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



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

- A [L<sup>A</sup>T<sub>E</sub>X 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.

## 1.4 Current Issues

### 1.4.1 Magit push in daemon

Quite often trying to push to a remote in the Emacs daemon produces an 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 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.

# CHAPTER 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 `~/ .authinfo.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 accidentally purge this file (I have done . 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 ellipsis are nicer than
↳ "...", and also save /precious/ space
password-cache-expiry nil ; I can trust my computers ... can't
↳ I?
;; scroll-preserve-screen-position 'always ; Don't have `point' jump around
scroll-margin 2) ; It's nice to maintain a little
↳ margin

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

(unless (string-match-p "^Power N/A" (battery)) ; On laptops...
  (display-battery-mode 1)) ; it's nice to know how much power
  ↳ you have

(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 `.after`. 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-vsplt-window-right t
      evil-split-window-below t)
```

Then, we'll pull up a buffer prompt.

```
(defadvice! prompt-for-buffer (&rest _)
  :after '(evil-window-split evil-window-vsplt)
  (consult-buffer))
```

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 through the `literate` module.

While we're in the `:config` section, we'll use Doooms 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)
(vertico +icons)       ; the search engine of the future
```

```
;;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
(emoji +unicode)      ; ζ
;;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)                 ; automated prettiness
;;god                    ; run Emacs commands without modifier keys
;;lisp                   ; 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                     ; 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
(:if (executable-find "aspell") spell) ; tasing you for misspelling misspelling
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...
;;beancount         ; mind the GAAP
;;cc                 ; C/C++/Obj-C madness
;;closure           ; 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
;;dhall             ; JSON with FP sprinkles
;;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
;;gdscrip              ; the language you waited for
;;(go +lsp)            ; the hipster dialect
;;(haskell +lsp)       ; 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 +lsp)           ; Python, R, and MATLAB in a blender
;;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
 +noter                ; enhanced PDF notetaking
 +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
 +roam2)               ; wander around notes
;;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
;;raku                 ; the artist formerly known as perl6
;;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
;;zig               ; C, but simpler
```

## Everything in Emacs

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

```
(:if (executable-find "mu") (mu4e +org +gmail))
;;notmuch
;;(wanderlust +gmail)
```

```
;;calendar          ; A dated approach to timetabling
;;emms              ; Multimedia in Emacs is music to my ears
everywhere          ; *leave* Emacs!? You must be joking.
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-unicode-font (font-spec :family "JuliaMono")
      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))
«
```

In addition to these fonts, Merriweather is used with `nov.el`, and Alegreya as a serified proportional font used by mixed-pitch-mode for `writeroom-mode` with Org files.

Because we care about how things look let's add a check to make sure we're told if the system doesn't have any of those fonts.

```
(defvar required-fonts '("JetBrainsMono.*" "Overpass" "JuliaMono" "IBM Plex Mono"
  ↪ "Merriweather" "Alegreya"))

(defvar available-fonts
  (delete-dups (or (font-family-list)
    (split-string (shell-command-to-string "fc-list : family"
      "[,\n]")))))

(defvar missing-fonts
  (delq nil (mapcar
    (lambda (font)
      (unless (delq nil (mapcar (lambda (f)
        (string-match-p (format "%s$" font) f))
          available-fonts))
        font))
    required-fonts)))

(if missing-fonts
  (pp-to-string
    `(unless noninteractive
      (add-hook! 'doom-init-ui-hook
        (run-at-time nil nil
          (lambda ()
            (message "%s missing the following fonts: %s"
              (propertize "Warning!" 'face '(bold warning))
              (mapconcat (lambda (font)
                (propertize font 'face
                  ↪ 'font-lock-variable-name-face))
                ',missing-fonts
                ", "))))
            (sleep-for 0.5))))))
```



```
;; No missing fonts detected")
```

```
<<detect-missing-fonts(>>>
```

This way whenever fonts are missing, after Doom's UI has initialised, a warning listing the missing fonts should appear for at least half a second.

## 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)
(remove-hook 'window-setup-hook #'doom-init-theme-h)
(add-hook 'after-init-hook #'doom-init-theme-h 'append)
(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 (and (memq (plist-get (coding-system-plist
    ↪ buffer-file-coding-system) :category)
      '(coding-category-undecided coding-category-utf-8))
      (not (memq (coding-system-eol-type
    ↪ buffer-file-coding-system) '(1 2))))
    t)))

(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")
```

### 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

### 2.3.4 Allow babel execution in CLI actions

In this config I sometimes generate code to include in my config. This works nicely, but for it to work with `doom sync` et. al. I need to make sure that Org doesn't try to confirm that I want to allow evaluation (I do!).

Thankfully Doom supports `$DOOMDIR/cli.el` file which is sourced every time a CLI command is run, so we can just enable evaluation by setting `org-confirm-babel-evaluate` to `nil` there. While we're at it, we should silence `org-babel-execute-src-block` to avoid polluting the output.

```
;;; cli.el -*- lexical-binding: t; -*-
(setq org-confirm-babel-evaluate nil)

(defun doom-shut-up-a (orig-fn &rest args)
  (quiet! (apply orig-fn args)))

(advice-add 'org-babel-execute-src-block :around #'doom-shut-up-a)
```

### 2.3.5 Emacs REPL

I think an emacs REPL sounds like a fun idea, even if not a particularly useful one. We can do this by adding a new command in `cli.el`.

```
(defcli! repl ((in-rlwrap-p ["--rl"] "For internal use only."))
  "Start an emacs REPL."
  (when (and (executable-find "rlwrap") (not in-rlwrap-p))
    ;; For autocomplete
    (setq autocomplete-file "/tmp/doom_elisp_repl_symbols")
    (unless (file-exists-p autocomplete-file)
      (princ "\e[0;33mInitialising autocomplete list...\e[0m\n")
      (with-temp-buffer
        (cl-do-all-symbols (s)
          (let ((sym (symbol-name s)))
            (when (string-match-p "\\`[:ascii:][:ascii:]+\\'" sym)
              (insert sym "\n"))))
        (write-region nil nil autocomplete-file)))
      (princ "\e[F")
      (throw 'exit (list "rlwrap" "-f" autocomplete-file
                        (concat doom-emacs-dir "bin/doom") "repl" "--rl"))))

  (doom-initialize-packages)
  (require 'engrave-faces-ansi)
  (setq engrave-faces-ansi-color-mode '3-bit)

  ;; For some reason (require 'parent-mode) doesn't work :(
  (defun parent-mode-list (mode)
    "Return a list of MODE and all its parent modes.
    The returned list starts with the parent-most mode and ends with MODE."
    (let ((result ()))
      (parent-mode--worker mode (lambda (mode)
                                   (push mode result)))
      result))
  (defun parent-mode--worker (mode func)
    "For MODE and all its parent modes, call FUNC.
    FUNC is first called for MODE, then for its parent, then for the parent's
    parent, and so on.
    MODE shall be a symbol referring to a function.
    FUNC shall be a function taking one argument."
    (funcall func mode)
    (when (not (fboundp mode))
      (signal 'void-function (list mode)))
    (let ((modefunc (symbol-function mode)))
      (if (symbolp modefunc)
          ;; Handle all the modes that use (defalias 'foo-parent-mode (stuff)) as
          ;; their parent
```

```

        (parent-mode--worker modefunc func)
      (let ((parentmode (get mode 'derived-mode-parent)))
        (when parentmode
          (parent-mode--worker parentmode func))))))
  (provide 'parent-mode)
;; Some extra highlighting (needs parent-mode)
(require 'rainbow-delimiters)
(require 'highlight-quoted)
(require 'highlight-numbers)
(setq emacs-lisp-mode-hook '(rainbow-delimiters-mode
                             highlight-quoted-mode
                             highlight-numbers-mode))

;; Pretty print
(defun pp-sexp (sexp)
  (with-temp-buffer
    (cl-prettyprint sexp)
    (emacs-lisp-mode)
    (font-lock-ensure)
    (with-current-buffer (engrave-faces-ansi-buffer)
      (princ (string-trim (buffer-string)))
      (kill-buffer (current-buffer)))))

;; Now do the REPL
(defvar accumulated-input nil)
(while t
  (condition-case nil
    (let ((input (if accumulated-input
                     (read-string "\e[31m .\e[0m ")
                     (read-string "\e[31mλ:\e[0m "))))
      (setq input (concat accumulated-input
                          (when accumulated-input "\n")
                          input)))

    (cond
     ((string-match-p "\\`[:space:]*\\'" input)
      nil)
     ((string= input "exit")
      (princ "\n") (kill-emacs 0))
     (t
      (condition-case err
        (let ((input-sexp (car (read-from-string input))))
          (setq accumulated-input nil)
          (pp-sexp (eval input-sexp))
          (princ "\n")))
        ;; Caused when sexp in unbalanced
        (end-of-file (setq accumulated-input input))
        (error
         (cl-destructuring-bind (backtrace &optional type data . _)
           (cons (doom-cli--backtrace) err)
           (princ (concat "\e[1;31mERROR:\e[0m " (get type 'error-message))))))
      ))))

```

```
(princ "\n
(pp-sexp (cons type data))
(when backtrace
  (print! (bold "Backtrace:"))
  (print-group!
    (dolist (frame (seq-take backtrace 10))
      (print!
        "%0.74s" (replace-regexp-in-string
                  "[\\n\\r]" "\\|\\n"
                  (format "%S" frame))))))
  (princ "\\n")))))))
;; C-d causes an end-of-file error
(end-of-file (princ "exit\\n") (kill-emacs 0)))
(unless accumulated-input (princ "\\n")))
```

### 2.3.6 Asynchronous config tangling

Doom adds an org-mode hook `+literate-enable-recompile-h`. This is a nice idea, but it's too blocking for my taste. Since I trust my tangling to be fairly straightforward, I'll just redefine it to a simpler, `async`, function.

```
(defvar +literate-tangle--proc nil)

(defvar +literate-tangle--proc-start-time nil)

(defadvice! +literate-tangle-async-h ()
  "A very simplified version of `+literate-tangle-h', but async."
  :override #' +literate-tangle-h
  (unless (getenv "__NOTANGLE")
    (let ((default-directory doom-private-dir))
      (when +literate-tangle--proc
        (message "Killing outdated tangle process...")
        (set-process-sentinel +literate-tangle--proc #'ignore)
        (kill-process +literate-tangle--proc)
        (sit-for 0.3)) ; ensure the message is seen for a bit
      (setq +literate-tangle--proc-start-time (float-time)
            +literate-tangle--proc
            (start-process "tangle-config"
                          (get-buffer-create " *tangle config*")
                          "emacs" "--batch" "--eval"))
```

```

(format "(progn \
  (require 'ox) \
  (require 'ob-tangle) \
  (setq org-confirm-babel-evaluate nil \
    org-inhibit-startup t \
    org-mode-hook nil \
    write-file-functions nil \
    before-save-hook nil \
    after-save-hook nil \
    vc-handled-backends nil \
    org-startup-folded nil \
    org-startup-indented nil) \
  (org-babel-tangle-file \"%s\" \"%s\")\"
  +literate-config-file
  (expand-file-name (concat doom-module-config-file
    ↪ ".el")))))

(set-process-sentinel +literate-tangle--proc #' +literate-tangle--sentinel)
(run-at-time nil nil (lambda () (message "Tangling config.org"))) ; ensure shown
↪ after a save message
"Tangling config.org..."))

(defun +literate-tangle--sentinel (process signal)
  (cond
    ((and (eq 'exit (process-status process))
      (= 0 (process-exit-status process)))
      (message "Tangled config.org sucessfully (took %.1fs)"
        (- (float-time) +literate-tangle--proc-start-time))
      (setq +literate-tangle--proc nil))
    ((memq (process-status process) (list 'exit 'signal))
      (+popup-buffer (get-buffer " *tangle config*")))
    (message "Failed to tangle config.org (after %.1fs)"
      (- (float-time) +literate-tangle--proc-start-time))
    (setq +literate-tangle--proc nil)))

(defun +literate-tangle-check-finished ()
  (when (and (process-live-p +literate-tangle--proc)
    (yes-or-no-p "Config is currently retangling, would you please wait a few
      ↪ seconds?"))
    (switch-to-buffer " *tangle config*")
    (signal 'quit nil)))
(add-hook! 'kill-emacs-hook #' +literate-tangle-check-finished)

```

### 2.3.7 Dashboard quick actions

When using the dashboard, there are often a small number of actions I will take. As the dashboard is it's own major mode, there is no need to suffer the tyranny of unnecessary keystrokes — we

can simply bind common actions to a single key!

```
(map! :map +doom-dashboard-mode-map
:ne "f" #'find-file
:ne "r" #'consult-recent-file
:ne "p" #'doom/open-private-config
:ne "c" (cmd! (find-file (expand-file-name "config.org" doom-private-dir)))
:ne "." (cmd! (doom-project-find-file "~/config/")) ; . for dotfiles
:ne "b" #'vertico/switch-workspace-buffer
:ne "B" #'consult-buffer
:ne "q" #'save-buffers-kill-terminal)
```

## 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*. I was using the blackhole image, but as I've stripped down the splash screen I've switched to just using the *E*.



Now we just make it theme-appropriate, and resize with the frame.

```
(defvar fancy-splash-image-template
  (expand-file-name "misc/splash-images/emacs-e-template.svg" doom-private-dir)
  "Default template svg used for the splash image, with substitutions from ")

(defvar fancy-splash-sizes
  `((:height 300 :min-height 50 :padding (0 . 2))
    (:height 250 :min-height 42 :padding (2 . 4))
    (:height 200 :min-height 35 :padding (3 . 3))
    (:height 150 :min-height 28 :padding (3 . 3))
    (:height 100 :min-height 20 :padding (2 . 2))
    (:height 75  :min-height 15 :padding (2 . 1))
    (:height 50  :min-height 10 :padding (1 . 0))
    (:height 1   :min-height 0  :padding (0 . 0)))
  "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' (top . bottom) 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 :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)
```

Now the only thing missing is a an extra interesting line, whether that be some corporate BS, an developer excuse, or a fun (useless) fact.

The following is rather long, but it essentially

- fetches a phrase from an API
- inserts it into the dashboard (asynchronously)
- moves point to the phrase
- re-uses the last phrase for requests within a few seconds of it being fetched

```
(defvar splash-phrases-source-folder
  (expand-file-name "misc/splash-phrases" doom-private-dir)
  "A folder of text files with a fun phrase on each line.")

(defvar splash-phrases-sources
  (let* ((files (directory-files splash-phrases-source-folder nil "\\..txt\\.."))
        (sets (delete-dups (mapcar
          (lambda (file)
            (replace-regexp-in-string "\\(?::-[0-9]+-\\w+\\)?\\.txt"
              ↪ "" file))
          files))))
    (mapcar (lambda (sset)
      (cons sset
        (delq nil (mapcar
          (lambda (file)
            (when (string-match-p (regexp-quote sset) file)
              file))
          files))))))
```

```

    sets))
  "A list of cons giving the phrase set name, and a list of files which contain phrase
  ↪ components.")

(defun splash-phrase-set
  (nth (random (length splash-phrase-sources)) (mapcar #'car splash-phrase-sources))
  "The default phrase set. See `splash-phrase-sources'.")

(defun splase-phrase-set-random-set ()
  "Set a new random splash phrase set."
  (interactive)
  (setq splash-phrase-set
    (nth (random (1- (length splash-phrase-sources)))
      (cl-set-difference (mapcar #'car splash-phrase-sources) (list
        ↪ splash-phrase-set))))
  (+doom-dashboard-reload t))

(defun splase-phrase--cache nil)

(defun splash-phrase-get-from-file (file)
  "Fetch a random line from FILE."
  (let ((lines (or (cdr (assoc file splase-phrase--cache))
    (cdar (push (cons file
      (with-temp-buffer
        (insert-file-contents (expand-file-name file
          ↪ splash-phrase-source-folder))
        (split-string (string-trim (buffer-string))
          ↪ "\n"))))
      splase-phrase--cache))))))
    (nth (random (length lines)) lines)))

(defun splash-phrase (&optional set)
  "Construct a splash phrase from SET. See `splash-phrase-sources'."
  (mapconcat
    #'splash-phrase-get-from-file
    (cdr (assoc (or set splash-phrase-set) splash-phrase-sources))
    " "))

(defun doom-dashboard-phrase ()
  "Get a splash phrase, flow it over multiple lines as needed, and make fontify it."
  (mapconcat
    (lambda (line)
      (+doom-dashboard--center
        +doom-dashboard--width
        (with-temp-buffer
          (insert-text-button
            line
            'action

```

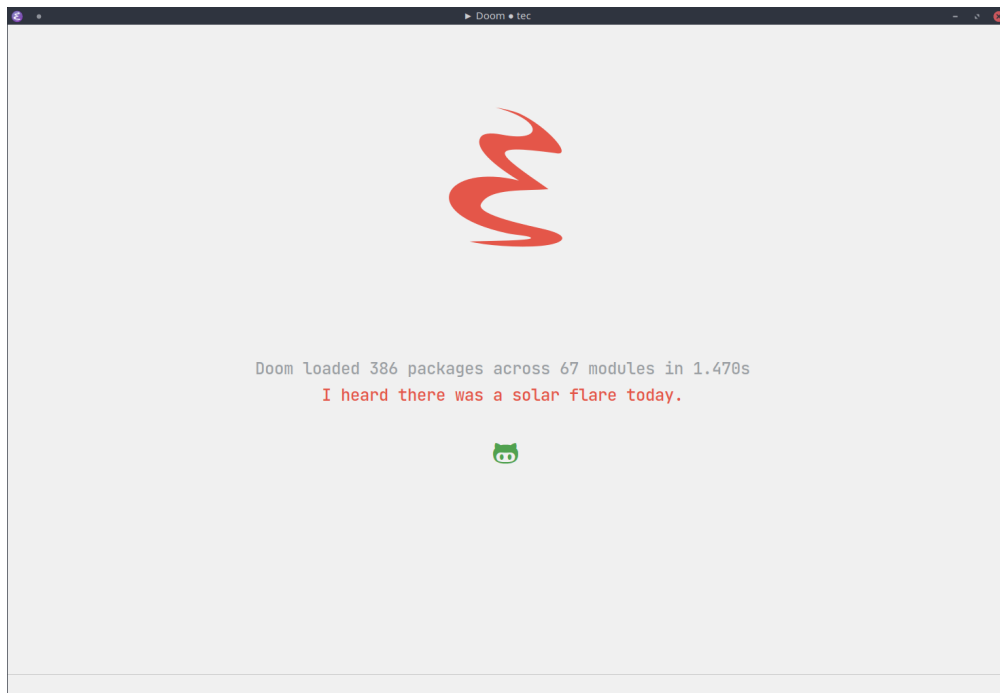
```
(lambda (_) (+doom-dashboard-reload t))
'face 'doom-dashboard-menu-title
'mouse-face 'doom-dashboard-menu-title
'help-echo "Random phrase"
'follow-link t)
(buffer-string)))
(split-string
 (with-temp-buffer
  (insert (splash-phrase))
  (setq fill-column (min 70 (/ (* 2 (window-width)) 3)))
  (fill-region (point-min) (point-max))
  (buffer-string))
 "\n")
 "\n"))

(defadvice! doom-dashboard-widget-loaded-with-phrase ()
:override #'doom-dashboard-widget-loaded
(setq line-spacing 0.2)
(insert
 "\n\n"
 (propertize
  (+doom-dashboard--center
   +doom-dashboard--width
   (doom-display-benchmark-h 'return))
  'face 'doom-dashboard-loaded)
 "\n"
 (doom-dashboard-phrase)
 "\n"))
```

Lastly, the doom dashboard “useful commands” are no longer useful to me. So, we’ll disable them and then for a particularly *clean* look disable the modeline and `hl-line-mode`, then also hide the cursor.

```
(remove-hook '+doom-dashboard-functions #'doom-dashboard-widget-shortmenu)
(add-hook! '+doom-dashboard-mode-hook (hide-mode-line-mode 1) (hl-line-mode -1))
(setq-hook! '+doom-dashboard-mode-hook evil-normal-state-cursor (list nil))
```

At the end, we have a minimal but rather nice splash screen.



I haven't forgotten about the ASCII banner though! Once again we're going for something simple.

```
(defun doom-dashboard-draw-ascii-emacs-banner ()
  (let* ((banner
          '(" ,---.,-.-.,---.,---.,---."
            "|---' | | |,---||      `---."
            "`----'`-  ' `---^`----'`-"))
        (longest-line (apply #'max (mapcar #'length banner))))
    (put-text-property
     (point)
     (dolist (line banner (point))
       (insert (+doom-dashboard--center
              +doom-dashboard--width
              (concat
               line (make-string (max 0 (- longest-line (length line)))
                                32)))
              "\n"))
     'face 'doom-dashboard-banner)))

(unless (display-graphic-p) ; for some reason this messes up the graphical splash
  => screen atm
  (setq +doom-dashboard-ascii-banner-fn #'doom-dashboard-draw-ascii-emacs-banner-fn))
```

## 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.

Lastly, while I'm not sure quite why it happens, but after a bit it seems that new Emacsclient frames start on the `*scratch*` buffer instead of the dashboard. I prefer the dashboard, so let's ensure that's always switched to in new frames.

```
(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)
  (add-hook! 'server-after-make-frame-hook
    (unless (string-match-p "\\*draft" (buffer-name))
      (switch-to-buffer +doom-dashboard-name))))
```

## 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 `ksh`

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'

```

## 2.4.6 Prompt to run setup script

At various points in this config, content is conditionally tangled to `./setup.sh`. It's no good just putting content there if it isn't run though. To help remind me to run it when needed, let's add a little prompt when there's anything to be run.

```

(if (file-exists-p "setup.sh")
    (if (string-empty-p (string-trim (with-temp-buffer (insert-file-contents
↳ "setup.sh") (buffer-string)) "#!/usr/bin/env bash"))
        (message ";; Setup script is empty")
        (message ";; Detected content in the setup script")
        (pp-to-string
            `(unless noninteractive
                (defun +config-run-setup ()

```

```
(when (yes-or-no-p (format "%s The setup script has content. Check and run
↳ the script?"
                                (propertize "Warning!" 'face '(bold warning))))
  (find-file (expand-file-name "setup.sh" doom-private-dir))
  (when (yes-or-no-p "Would you like to run this script?")
    (async-shell-command "./setup.sh"))))
(add-hook! 'doom-init-ui-hook
  (run-at-time nil nil #' +config-run-setup))))
(message ";; setup.sh did not exist during tangle. Tangle again.")
(pp-to-string
  `(unless noninteractive
    (add-hook! 'doom-init-ui-hook #' +iterate-tangle-async-h))))
```

```
<<run-setup()>>
```

## CHAPTER Packages 3

### 3.1 Loading instructions

This is where you install packages, by declaring them with the `package!` macro in `packages.el`, then running `doom refresh` on the command line. This file shouldn't be byte compiled.

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

You'll then 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.

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

## 3.2 Convenience

### 3.2.1 Avy

*From the `:config` default module.*

What a wonderful way to jump to buffer positions, and it uses the QWERTY home-row for jumping. Very convenient ... except I'm using Colemak.

```
(after! avy  
  ;; home row priorities: 8 6 4 5 - - 1 2 3 7  
  (setq avy-keys '(?n ?e ?i ?s ?t ?r ?i ?a)))
```

### 3.2.2 Rotate (window management)

The rotate package just adds the ability to rotate window layouts, but that sounds nice to me.

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

### 3.2.3 Emacs Everywhere

The name says it all. It's loaded and set up (a bit) by `:app everywhere`, however as I develop this I want the unpinned version I have as a submodule.

```
(package! emacs-everywhere :recipe (:local-repo "lisp/emacs-everywhere"))
(unpin! emacs-everywhere)
```

Additionally, I'm going to make some personal choices that aren't made in the Doom module.

```
(use-package! emacs-everywhere
  :if (daemonp)
  :config
  (require 'spell-fu)
  (setq emacs-everywhere-major-mode-function #'org-mode
        emacs-everywhere-frame-name-format "Edit ⚡ %s - %s")
  (defadvice! emacs-everywhere-raise-frame ()
    :after #'emacs-everywhere-set-frame-name
    (setq emacs-everywhere-frame-name (format emacs-everywhere-frame-name-format
                                              (emacs-everywhere-app-class
                                               ↪ emacs-everywhere-current-app)
                                              (truncate-string-to-width
                                               (emacs-everywhere-app-title
                                                ↪ emacs-everywhere-current-app)
                                               45 nil nil "..."))))
  ;; need to wait till frame refresh happen before really set
  (run-with-timer 0.1 nil #'emacs-everywhere-raise-frame-1))
  (defun emacs-everywhere-raise-frame-1 ()
    (call-process "wmctrl" nil nil nil "-a" emacs-everywhere-frame-name)))
```

### 3.2.4 Which-key

*From the `:core packages` module.*

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]
```

## 3.3 Tools

### 3.3.1 Abbrev

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

```
(add-hook 'doom-first-buffer-hook
  (defun +abbrev-file-name ()
    (setq-default abbrev-mode t)
    (setq abbrev-file-name (expand-file-name "abbrev.el" doom-private-dir))))
```

### 3.3.2 Very 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)
```

### 3.3.3 Eros

*From the `:tools eval` module.*

This package enables the very nice inline evaluation with `gr` and `gR`. The prefix could be slightly nicer though.

```
(setq eros-eval-result-prefix "¿ ") ; default =>
```

### 3.3.4 EVIL

*From the `:editor evil` module.*

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

Now, `EVIL` cares a fair bit about keeping compatibility with Vim's default behaviour. I don't. There are some particular settings that I'd rather be something else, so let's change them.

```
(after! evil
  (setq evil-ex-substitute-global t      ; I like my s/./.. to be global by default
        evil-move-cursor-back nil      ; Don't move the block cursor when toggling
        ↷ insert mode
        evil-kill-on-visual-paste nil)) ; Don't put overwritten text in the kill ring
```

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... It seems that there's a dedicated package for this, so instead of just disabling the mode on startup, let's prevent installation of the package.

```
(package! evil-escape :disable t)
```

### 3.3.5 Consult

*From the `:completion vertico` module.*

Since we're using Section 3.4.9 too, the separation between buffers and files is already clear, and there's no need for a different face.

```
(after! consult
  (set-face-attribute 'consult-file nil :inherit 'consult-buffer)
  (setf (plist-get (alist-get 'perl consult-async-split-styles-alist) :initial) ";"))
```

### 3.3.6 Magit

From the `:tools magit` module.



**Git** If that doesn't fix it, `git.txt` contains the phone number of a friend of mine who understands git. Just wait through a few minutes of 'It's really pretty simple, just think of branches as...' and eventually you'll learn the commands that will fix everything.

Magit is great as-is, thanks for making such a lovely package [Jonas](#)!

### Commit message templates

One little thing I want to add is some per-project commit message templates.

```
(defvar +magit-project-commit-templates-alist nil
  "Alist of toplevel dirs and template strings/functions.")
(after! magit
  (defun +magit-fill-in-commit-template ()
    "Insert template from `+magit-fill-in-commit-template' if applicable."
    (when-let ((template (cdr (assoc (file-name-base (directory-file-name
      ↪ (magit-toplevel))))
      +magit-project-commit-templates-alist))))
      (goto-char (point-min))
      (insert (if (stringp template) template (funcall template))))
```



```
(goto-char (point-min))
(end-of-line)))
(add-hook 'git-commit-setup-hook #'magit-fill-in-commit-template 90))
```

This is particularly useful when creating commits for Org, as they need to follow [a certain format](#) and sometimes I forget elements (oops!).

```
(after! magit
  (defun +org-commit-message-template ()
    "Create a skeleton for an Org commit message based on the staged diff."
    (let (change-data last-file file-changes temp-point)
      (with-temp-buffer
        (apply #'call-process magit-git-executable
          nil t nil
          (append
            magit-git-global-arguments
            (list "diff" "--cached"))))
        (goto-char (point-min))
        (message "%s" (buffer-string))
        (while (re-search-forward "@@\\|\\|+\\|+\\|+ b/" nil t)
          (if (looking-back "\\|+\\|+ b/" (line-beginning-position))
              (progn
                (push (list last-file file-changes) change-data)
                (setq last-file (buffer-substring-no-properties (point)
                  ⇒ (line-end-position)
                  file-changes nil))
                (setq temp-point (line-beginning-position))
                (re-search-forward "\\|+\\|+^-" nil t)
                (end-of-line))
              (cond
                ((string-match-p "\\..el$" last-file)
                 (when (re-search-backward "\\(?:[:+]? *\\|@[@ +-\\|d,]+@@"
                  ⇒ "\\(?:cl-\\|\\(?:defun\\|defvar\\|defmacro\\|defcustom\\|)"
                  ⇒ temp-point t)
                   (re-search-forward
                     ⇒ "\\(?:cl-\\|\\(?:defun\\|defvar\\|defmacro\\|defcustom\\|)" nil
                     ⇒ t)
                   (add-to-list 'file-changes (buffer-substring-no-properties (point)
                     ⇒ (forward-symbol 1))))))
                ((string-match-p "\\..org$" last-file)
                 (when (re-search-backward "[+-]\\|+ \\|+^@[@ +-\\|d,]+@@"
                  ⇒ temp-point t)
                   (re-search-forward "@@" nil t)
                   (add-to-list 'file-changes (buffer-substring-no-properties (point)
                     ⇒ (line-end-position))))))))
        (push (list last-file file-changes) change-data)
        (setq change-data (delete '(nil nil) change-data))
        (concat
```

```

(if (= 1 (length change-data))
  (replace-regexp-in-string "^.*?\\|\\. [a-z]+$" "" (caar change-data))
  "?")
": \n\n"
(mapconcat
  (lambda (file-changes)
    (if (cadr file-changes)
      (format "%s (%s): "
              (car file-changes)
              (mapconcat #'identity (cadr file-changes) ", "))
      (format "%s: " (car file-changes))))
  change-data
  "\n\n"))

(add-to-list '+magit-project-commit-templates-alist (cons "org-mode"
  ↪ #' +org-commit-message-template)))

```

This relies on two small entries in the git config files which improves the hunk heading line selection for elisp and Org files.

```

[diff "lisp"]
xfuncname = "^(((;+ )|\\(|([
  ↪ \t])+\\(((c\\l-|e\\l-patch-)?def(un|var|macro|method|custom)|gb/)))\\.*)$"

[diff "org"]
xfuncname = "^((\\*+ +\\.*)$"

```

### 3.3.7 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 (which don't get any syntax highlighting by default). This requires the delta binary. It's packaged on some distributions, but most reliably installed through Rust's package manager cargo.

```
cargo install git-delta
```

Now we can make use of the package for this.

```

;; (package! magit-delta :recipe (:host github :repo "dandavison/magit-delta") :pin
  ↪ "56cdfd377279589aa0cb1df99455c098f1848cf")

```

All that's left is to hook it into magit

```

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

```

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

### 3.3.8 Smerge

For repeated operations, a hydra would be helpful. But I prefer transient.

```
(defun smerge-repeatedly ()
  "Perform smerge actions again and again"
  (interactive)
  (smerge-mode 1)
  (smerge-transient))
(after! transient
 (transient-define-prefix smerge-transient ()
  ["Move"
   ("n" "next" (lambda () (interactive) (ignore-errors (smerge-next))
    ↪ (smerge-repeatedly)))
   ("p" "previous" (lambda () (interactive) (ignore-errors (smerge-prev))
    ↪ (smerge-repeatedly)))]
  ["Keep"
   ("b" "base" (lambda () (interactive) (ignore-errors (smerge-keep-base))
    ↪ (smerge-repeatedly)))
   ("u" "upper" (lambda () (interactive) (ignore-errors (smerge-keep-upper))
    ↪ (smerge-repeatedly)))
   ("l" "lower" (lambda () (interactive) (ignore-errors (smerge-keep-lower))
    ↪ (smerge-repeatedly)))
   ("a" "all" (lambda () (interactive) (ignore-errors (smerge-keep-all))
    ↪ (smerge-repeatedly)))
   ("RET" "current" (lambda () (interactive) (ignore-errors (smerge-keep-current))
    ↪ (smerge-repeatedly)))]
  ["Diff"
   ("<" "upper/base" (lambda () (interactive) (ignore-errors
    ↪ (smerge-diff-base-upper)) (smerge-repeatedly)))
   ("=" "upper/lower" (lambda () (interactive) (ignore-errors
    ↪ (smerge-diff-upper-lower)) (smerge-repeatedly)))
   (">" "base/lower" (lambda () (interactive) (ignore-errors
    ↪ (smerge-diff-base-lower)) (smerge-repeatedly)))
   ("R" "refine" (lambda () (interactive) (ignore-errors (smerge-refine))
    ↪ (smerge-repeatedly)))
   ("E" "ediff" (lambda () (interactive) (ignore-errors (smerge-ediff))
    ↪ (smerge-repeatedly)))]
  ["Other"
   ("c" "combine" (lambda () (interactive) (ignore-errors
    ↪ (smerge-combine-with-next)) (smerge-repeatedly)))
   ("r" "resolve" (lambda () (interactive) (ignore-errors (smerge-resolve))
    ↪ (smerge-repeatedly)))])
```

```
("k" "kill current" (lambda () (interactive) (ignore-errors  
  ↪ (smerge-kill-current)) (smerge-repeatedly)))  
("q" "quit" (lambda () (interactive) (smerge-auto-leave))))))
```

### 3.3.9 Company

*From the `:completion` company module.*

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)
```

### 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.

**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))
```

**3.3.10 Projectile**

*From the :core packages module.*

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)))
```

**3.3.11 Ispell****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** o

**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&specia
↳ 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&special=
  ↪ numerals&encoding=utf-8&format=inline&download=aspell'
tar -xjf "aspell6-en-custom.tar.bz2"

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

## 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))
```

### 3.3.12 TRAMP

Another lovely Emacs feature, TRAMP stands for *Transparent Remote Access, Multiple Protocol*. In brief, it's a lovely way to wander around outside your local filesystem.

## Prompt recognition

Unfortunately, when connecting to remote machines Tramp can be a wee bit picky with the prompt format. Let's try to get Bash, and be a bit more permissive with prompt recognition.

```
(after! tramp
  (setenv "SHELL" "/bin/bash")
  (setq tramp-shell-prompt-pattern "\\(?:^\\|
    \\)[^#%>\\n]*#?[]#%>[] *\\(\\[[0-9;]*[a-zA-Z] *\\)*")) ;; default + 2
```

## 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
```

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

### 3.3.13 Auto activating snippets

Sometimes pressing **TAB** is just too much.

```
(package! aas :recipe (:host github :repo "ymarco/auto-activating-snippets")
  :pin "1699bec4d244a1f62af29fe4eb8b79b6d2fccf7d")
```

```
(use-package! aas
  :commands aas-mode)
```

### 3.3.14 Screenshot

This makes it a breeze to take lovely screenshots.

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

#### ● Screenshots

This makes it a breeze to take lovely screenshots.

```
» emacs-lisp
```

```
(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"))
```

### 3.3.15 Etrace

The *Emacs Lisp Profiler* (ELP) does a nice job recording information, but it isn't the best for looking at results. etrace converts ELP's results to the "Chromium Catapult Trace Event Format". This means that the output of etrace can be loaded in something like the [speedscope](#) webapp for easier profile investigation.

```
(package! etrace :recipe (:host github :repo "aspier/etrace"))
```

```
(use-package! etrace
  :after elp)
```

### 3.3.16 YASnippet

*From the :editor snippets module.*



Nested snippets are good, so let's enable that.

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

### 3.3.17 String inflection

For when you want to change the case pattern for a symbol.

```
(package! string-inflection :pin "fd7926ac17293e9124b31f706a4e8f38f6a9b855")
```

```
(use-package! string-inflection
  :commands (string-inflection-all-cycle
              string-inflection-toggle
              string-inflection-camelcase
              string-inflection-lower-camelcase
              string-inflection-kebab-case
              string-inflection-underscore
              string-inflection-capital-underscore
              string-inflection-upcase)

  :init
  (map! :leader :prefix ("c~" . "naming convention")
        :desc "cycle" "~" #'string-inflection-all-cycle
        :desc "toggle" "t" #'string-inflection-toggle
        :desc "CamelCase" "c" #'string-inflection-camelcase
        :desc "downCase" "d" #'string-inflection-lower-camelcase
        :desc "kebab-case" "k" #'string-inflection-kebab-case
        :desc "under_score" "_" #'string-inflection-underscore
        :desc "Upper_Score" "u" #'string-inflection-capital-underscore
        :desc "UP_CASE" "U" #'string-inflection-upcase)

  (after! evil
    (evil-define-operator evil-operator-string-inflection (beg end _type)
      "Define a new evil operator that cycles symbol casing."
      :move-point nil
      (interactive "<R>")
      (string-inflection-all-cycle)
      (setq evil-repeat-info '([?g ?~])))
    (define-key evil-normal-state-map (kbd "g~") 'evil-operator-string-inflection)))
```

### 3.3.18 Smart parentheses

*From the :core packages module.*

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

## 3.4 Visuals

### 3.4.1 Info colours

This makes manual pages nicer to look at by adding variable pitch fontification and colouring ꞑ.

## 2.9 Set operations

Operations pretending lists are sets.

-- **Function:** `-union (list list2)`  
 Return a new list containing the elements of LIST and elements of LIST2 that are not in LIST. The test for equality is done with 'equal', or with '-compare-fn' if that's non-nil.

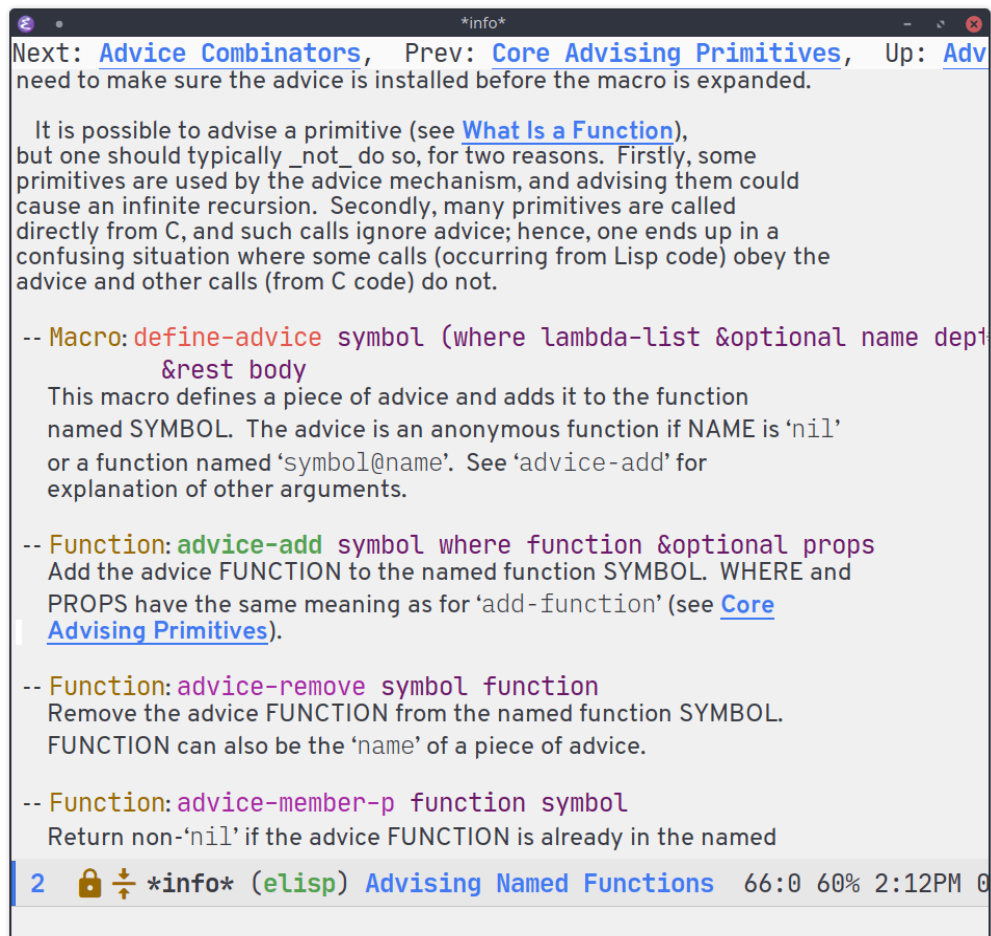
```
(-union '(1 2 3) '(3 4 5))
⇒ '(1 2 3 4 5)
(-union '(1 2 3 4) '())
⇒ '(1 2 3 4)
(-union '(1 1 2 2) '(3 2 1))
⇒ '(1 1 2 2 3)
```

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

To use this we'll just hook it into Info.

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

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



### 3.4.2 Modus themes

Proteolas did a lovely job with the Modus themes, so much so that they were welcomed into Emacs 28. However, he is also rather attentive with updates, and so I'd like to make sure we have a recent version.

```
(package! modus-themes :pin "392ebb115b07f8052d512ec847619387d109edd6")
```

### 3.4.3 Theme magic

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

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

This operates using `pywal`, which is present in some repositories, but most reliably installed with `pip`.

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

Theme magic takes a look at a number of faces, the saturation levels, and colour differences to try to cleverly pick eight colours to use. However, it uses the same colours for the light variants, and doesn't always make the best picks. Since we're using doom-themes, our life is a little easier and we can use the colour utilities from Doom themes to easily grab sensible colours and generate lightened versions — let's do that.

```
(use-package! theme-magic
  :commands theme-magic-from-emacs
  :config
  (defadvice! theme-magic--auto-extract-16-doom-colors ()
    :override #'theme-magic--auto-extract-16-colors
    (list
      (face-attribute 'default :background)
      (doom-color 'error)
      (doom-color 'success)
      (doom-color 'type)
      (doom-color 'keywords)
      (doom-color 'constants)
      (doom-color 'functions)
      (face-attribute 'default :foreground)
      (face-attribute 'shadow :foreground)
      (doom-blend 'base8 'error 0.1)
      (doom-blend 'base8 'success 0.1)
      (doom-blend 'base8 'type 0.1)
      (doom-blend 'base8 'keywords 0.1)
      (doom-blend 'base8 'constants 0.1)
      (doom-blend 'base8 'functions 0.1)
      (face-attribute 'default :foreground))))
```

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)))
```

### 3.4.4 Emojify

*From the `:ui emoji` module.*

For starters, twitter's emojis look nicer than emoji-one. Other than that, this is pretty great OOTB.

```
(setq emojiify-emoji-set "twemoji-v2")
```

One minor annoyance is the use of emojis over the default character when the default is actually preferred. This occurs with overlay symbols I use in Org mode, such as checkbox state, and a few other miscellaneous cases.

We can accommodate our preferences by deleting those entries from the emoji hash table

```
(defvar emojiify-disabled-emojis
  ' (; Org
    "⌘" "⌘" "⌘" "⌘" "⌘" "⌘" "⌘" "⌘" "⌘"
    ;; Terminal powerline
    "⌘"
    ;; Box drawing
    "⌘" "⌘")
  "Characters that should never be affected by `emojiify-mode'.")

(defadvice! emojiify-delete-from-data ()
  "Ensure `emojiify-disabled-emojis' don't appear in `emojiify-emojis'."
  :after #'emojiify-set-emoji-data
  (dolist (emoji emojiify-disabled-emojis)
    (remhash emoji emojiify-emojis)))
```

Now, it would be good to have a minor mode which allowed you to type ascii/gh emojis and get them converted to unicode. Let's make one.

```
(defun emojiify--replace-text-with-emoji (orig-fn emoji text buffer start end &optional
  ↪ target)
  "Modify `emojiify--propertize-text-for-emoji' to replace ascii/github emoticons with
  ↪ unicode emojis, on the fly."
  (if (or (not emoticon-to-emoji) (= 1 (length text)))
      (funcall orig-fn emoji text buffer start end target)
      (delete-region start end)
      (insert (ht-get emoji "unicode"))))

(define-minor-mode emoticon-to-emoji
  "Write ascii/gh emojis, and have them converted to unicode live."
  :global nil
  :init-value nil
  (if emoticon-to-emoji
```

```
(progn
  (setq-local emojiify-emoji-styles '(ascii github unicode))
  (advice-add 'emojiify--propertize-text-for-emoji :around
    ↪ #'emojiify--replace-text-with-emoji)
  (unless emojiify-mode
    (emojiify-turn-on-emojiify-mode)))
(setq-local emojiify-emoji-styles (default-value 'emojiify-emoji-styles))
(advice-remove 'emojiify--propertize-text-for-emoji
  ↪ #'emojiify--replace-text-with-emoji))
```

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)
  ↪ (emojiify-mode 1))
```

### 3.4.5 Doom modeline

*From the :ui modeline module.*

Very nice and pretty, however I think the PDF modeline could do with tweaking. I raised [an issue](#) on this, however the response was basically “put your preferences in your personal config, the current default is sensible” — so here we are.

First up I'm going to want a segment for just the buffer file name, and a PDF icon. Then we'll redefine two functions used to generate the modeline.

```
(after! doom-modeline
  (doom-modeline-def-segment buffer-name
    "Display the current buffer's name, without any other information."
    (concat
      (doom-modeline-spc)
      (doom-modeline--buffer-name)))

  (doom-modeline-def-segment pdf-icon
    "PDF icon from all-the-icons."
    (concat
      (doom-modeline-spc)
      (doom-modeline-icon 'octicon "file-pdf" nil nil
        :face (if (doom-modeline--active)
          'all-the-icons-red
          'mode-line-inactive)
        :v-adjust 0.02)))

  (defun doom-modeline-update-pdf-pages ()
    "Update PDF pages."
```

```
(setq doom-modeline--pdf-pages
  (let ((current-page-str (number-to-string (eval `(pdf-view-current-page))))
        (total-page-str (number-to-string (pdf-cache-number-of-pages))))
    (concat
      (propertize
        (concat (make-string (- (length total-page-str) (length
          ↪ current-page-str)) ? )
          " P" current-page-str)
        'face 'mode-line)
      (propertize (concat "/" total-page-str) 'face
        ↪ 'doom-modeline-buffer-minor-mode))))))

(doom-modeline-def-segment pdf-pages
  "Display PDF pages."
  (if (doom-modeline--active) doom-modeline--pdf-pages
      (propertize doom-modeline--pdf-pages 'face 'mode-line-inactive)))

(doom-modeline-def-modeline 'pdf
  '(bar window-number pdf-pages pdf-icon buffer-name)
  '(misc-info matches major-mode process vcs))
```

### 3.4.6 Keycast

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.

```
2 ~/.config/doom/ SPC SPC +ivy/projectile-find-file 4:32PM 1.46 DOOM v2.0.9
```

```
(package! keycast :pin "04ba7519f34421c235bac458f0192c130f732f12")
```

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)))
```

### 3.4.7 Screenshot

In a similar manner to Section 3.4.6, [gif-screenshot](#) may come in handy.

```
(package! gif-screenshot :pin "5517a557a17d8016c9e26b0acb74197550f829b9")
```

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 fix, 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-screenshot
  :commands gif-screenshot-mode
  :config
  (map! :map gif-screenshot-mode-map
    :g "<f8>" #'gif-screenshot-toggle-pause
    :g "<f9>" #'gif-screenshot-stop)
  (setq gif-screenshot-program "maim"
        gif-screenshot-args `("--quality" "3" "-i" ,(string-trim-right
          (shell-command-to-string
            "xdotool getactivewindow")))
        gif-screenshot-optimize-args '("--batch" "--optimize=3"
          ↳ "--usecolormap=/tmp/doom-color-theme"))
  (defun gif-screenshot-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))

```

### 3.4.8 Mixed pitch

*From the :ui zen module.*

We'd like to use mixed pitch in certain modes. If we simply add a hook, when directly opening a file with (a new) Emacs mixed-pitch-mode runs before UI initialisation, which is problematic. To resolve this, we create a hook that runs after UI initialisation and both

- conditionally enables mixed-pitch-mode
- sets up the mixed pitch hooks

```

(defvar mixed-pitch-modes '(org-mode LaTeX-mode markdown-mode gfm-mode Info-mode)
  "Modes that mixed-pitch-mode should be enabled in, but only after UI
  ↪ initialisation.")
(defun init-mixed-pitch-h ()
  "Hook mixed-pitch-mode into each mode in mixed-pitch-modes.
  Also immediately enables mixed-pitch-modes if currently in one of the modes."
  (when (memq major-mode mixed-pitch-modes)
    (mixed-pitch-mode 1))
  (dolist (hook mixed-pitch-modes)
    (add-hook (intern (concat (symbol-name hook) "-hook")) #'mixed-pitch-mode)))
  (add-hook 'doom-init-ui-hook #'init-mixed-pitch-h))

```

As mixed pitch uses the variable `mixed-pitch-face`, we can create a new function to apply mixed pitch with a serif face instead of the default. This was created for `writeroom` mode.

```

(autoload #'mixed-pitch-serif-mode "mixed-pitch"
  "Change the default face of the current buffer to a serified variable pitch, while
  ↪ keeping some faces fixed pitch." t)

(after! mixed-pitch
  (defface variable-pitch-serif
    '((t (:family "serif")))
    "A variable-pitch face with serifs."
    :group 'basic-faces)
  (setq mixed-pitch-set-height t)
  (setq variable-pitch-serif-font (font-spec :family "Alegreya" :size 27))
  (set-face-attribute 'variable-pitch-serif nil :font variable-pitch-serif-font)
  (defun mixed-pitch-serif-mode (&optional arg)

```

```
"Change the default face of the current buffer to a serified variable pitch, while
↳ keeping some faces fixed pitch."
(interactive)
(let ((mixed-pitch-face 'variable-pitch-serif))
  (mixed-pitch-mode (or arg 'toggle))))
```

Now, as Harfbuzz is currently used in Emacs, we'll be missing out on the following Alegreya ligatures:

ff ff fi fi ff fi ff fi ff fi fi fi fi fi fi Th Th

Thankfully, it isn't too hard to add these to the composition-function-table.

```
(set-char-table-range composition-function-table ?f '(["\\(?:ff?[fijlt]\\)" 0
↳ font-shape-gstring]))
(set-char-table-range composition-function-table ?T '(["\\(?:Th\\)" 0
↳ font-shape-gstring]))
```

### 3.4.9 Marginalia

*Part of the :completion vertico module.*

Marginalia is nice, but the file metadata annotations are a little too plain. Specifically, I have these gripes

- File attributes would be nicer if coloured
- I don't care about the user/group information if the user/group is me
- When a file time is recent, a relative age (e.g. 2h ago) is more useful than the date
- An indication of file fatness would be nice

Thanks to the marginalia-annotator-registry, we don't have to advise, we can just add a new file annotator.

Another small thing is the face used for docstrings. At the moment it's (*italic shadow*), but I don't like that.

```
(after! marginalia
  (setq marginalia-censor-variables nil)

  (defadvice! +marginalia--annotate-local-file-colorful (cand)
    "Just a more colourful version of `marginalia--annotate-local-file'."
    :override #'marginalia--annotate-local-file)
```

```

(when-let (attrs (file-attributes (substitute-in-file-name
                                  (marginalia--full-candidate cand))
                                  'integer))

  (marginalia--fields
    ((marginalia--file-owner attrs)
     :width 12 :face 'marginalia-file-owner)
    ((marginalia--file-modes attrs))
    ((+marginalia-file-size-colorful (file-attribute-size attrs))
     :width 7)
    ((+marginalia--time-colorful (file-attribute-modification-time attrs))
     :width 12))))

(defun +marginalia--time-colorful (time)
  (let* ((seconds (float-time (time-subtract (current-time) time)))
        (color (doom-blend
                  (face-attribute 'marginalia-date :foreground nil t)
                  (face-attribute 'marginalia-documentation :foreground nil t)
                  (/ 1.0 (log (+ 3 (/ (+ 1 seconds) 345600.0))))))
    ;; 1 - log(3 + 1/(days + 1)) % grey
    (propertize (marginalia--time time) 'face (list :foreground color))))

(defun +marginalia-file-size-colorful (size)
  (let* ((size-index (/ (log10 (+ 1 size)) 7.0))
        (color (if (< size-index 10000000) ; 10m
                    (doom-blend 'orange 'green size-index)
                    (doom-blend 'red 'orange (- size-index 1)))))
    (propertize (file-size-human-readable size) 'face (list :foreground color))))

```

### 3.4.10 Centaur Tabs

*From the :ui tabs module.*

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 `~\_{(i)}~`. 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 "x"
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)
```

### 3.4.11 All the icons

*From the :core packages module.*

all-the-icons does a generally great job giving file names icons. One minor niggle I have is that when I open a .m file, it's much more likely to be Matlab than Objective-C. As such, it'll be switching the icon associated with .m.

```
(after! all-the-icons
  (setcdr (assoc "m" all-the-icons-extension-icon-alist)
    (cdr (assoc "matlab" all-the-icons-extension-icon-alist))))
```

### 3.4.12 Prettier page breaks

In some files, ^L appears as a page break character. This isn't that visually appealing, and Steve Purcell has been nice enough to make a package to display these as horizontal rules.

```
(package! page-break-lines :recipe (:host github :repo "purcell/page-break-lines"))
```

```
(use-package! page-break-lines
  :commands page-break-lines-mode
  :init
  (autoload 'turn-on-page-break-lines-mode "page-break-lines")
  :config
  (setq page-break-lines-max-width fill-column)
  (map! :prefix "g"
    :desc "Prev page break" :nv "[" #'backward-page
    :desc "Next page break" :nv "]" #'forward-page))
```

### 3.4.13 Writeroom

*From the :ui zen module.*

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

```
(setq +zen-text-scale 0.8)
```

Then, when using Org it would be nice to make a number of other aesthetic tweaks. Namely:

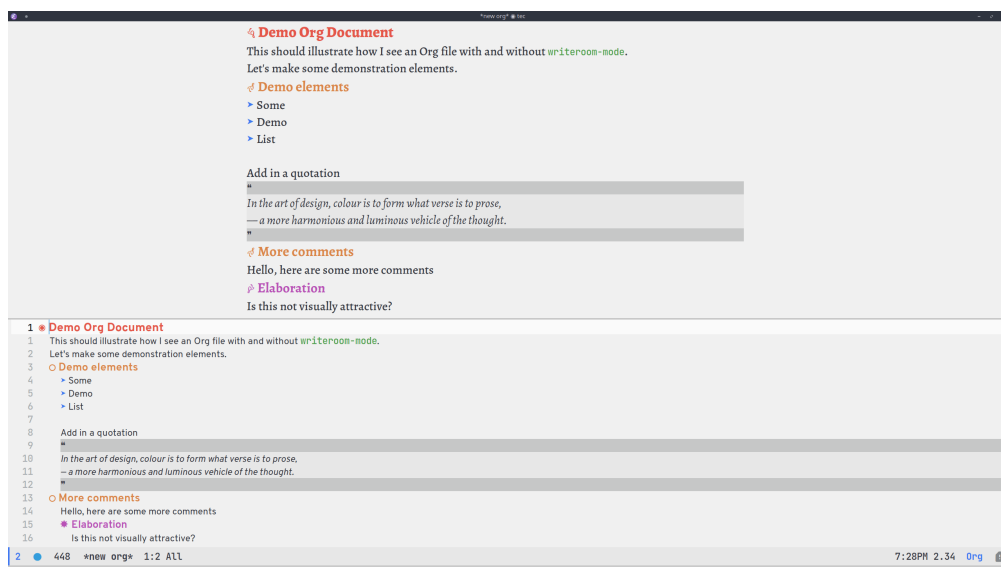
- Use a serified variable-pitch font
- Hiding headline leading stars
- Using fleurons as headline bullets
- Hiding line numbers
- Removing outline indentation
- Centring the text
- Turning on org-pretty-table-mode

```
(defvar +zen-serif-p t
  "Whether to use a serified font with `mixed-pitch-mode'.")
(after! writeroom-mode
  (defvar-local +zen--original-org-indent-mode-p nil)
  (defvar-local +zen--original-mixed-pitch-mode-p nil)
  (defvar-local +zen--original-org-pretty-table-mode-p nil)
  (defun +zen-enable-mixed-pitch-mode-h ()
    "Enable `mixed-pitch-mode' when in `+zen-mixed-pitch-modes'."
    (when (apply #'derived-mode-p +zen-mixed-pitch-modes)
      (if writeroom-mode
        (progn
          (setq +zen--original-mixed-pitch-mode-p mixed-pitch-mode)
          (funcall (if +zen-serif-p #'mixed-pitch-serif-mode #'mixed-pitch-mode) 1))
        (funcall #'mixed-pitch-mode (if +zen--original-mixed-pitch-mode-p 1 -1)))))
  (pushnew! writeroom--local-variables
    'display-line-numbers
    'visual-fill-column-width
    'org-adapt-indentation
    'org-superstar-headline-bullets-list
    'org-superstar-remove-leading-stars)
  (add-hook 'writeroom-mode-enable-hook
    (defun +zen-prose-org-h ()
      "Reformat the current Org buffer appearance for prose."
      (when (eq major-mode 'org-mode)
        (setq display-line-numbers nil
              visual-fill-column-width 60
              org-adapt-indentation nil)
        (when (featurep 'org-superstar)
          (setq-local org-superstar-headline-bullets-list '("¿" "¿" "¿" "¿")
                    ;; org-superstar-headline-bullets-list '("¿" "¿" "¿" "¿"
                    ↪ "¿" "¿" "¿" "¿")
                    org-superstar-remove-leading-stars t))
```

```

(org-superstar-restart))
(setq
 +zen--original-org-indent-mode-p org-indent-mode
 +zen--original-org-pretty-table-mode-p (bound-and-true-p
 ↪ org-pretty-table-mode))
(org-indent-mode -1)
(org-pretty-table-mode 1)))
(add-hook 'writeroom-mode-disable-hook
 (defun +zen-nonprose-org-h ()
  "Reverse the effect of `+zen-prose-org'."
  (when (eq major-mode 'org-mode)
    (when (featurep 'org-superstar)
      (org-superstar-restart))
    (when +zen--original-org-indent-mode-p (org-indent-mode 1))
    ;; (unless +zen--original-org-pretty-table-mode-p
    ↪ (org-pretty-table-mode -1))
    )))

```



### 3.4.14 Treemacs

From the `:ui treemacs` module.

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 be 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
    ↪ `treemacs-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"))

```

## 3.5 Frivolities

### 3.5.1 xkcd

XKCD comics are fun.

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

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 `completing-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 (completing-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 completing-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 than
  ↪ `+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 "^$"))
              (progn
                (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))
                        ))))
                    ))))
```

### 3.5.2 Selectric

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")
```

```
(use-package! selectric-mode
  :commands selectric-mode)
```

### 3.5.3 Wttrin

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/wttrin"))
```

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

### 3.5.4 Spray

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

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

It would be nice if Spray's default speed suited me better, and the keybindings worked in evil mode. Let's do that and make the display slightly nicer while we're at it.

```
(use-package! spray
  :commands spray-mode
  :config
  (setq spray-wpm 600
        spray-height 800)
  (defun spray-mode-hide-cursor ()
    "Hide or unhide the cursor as is appropriate."
    (if spray-mode
        (setq-local spray--last-evil-cursor-state evil-normal-state-cursor
                     evil-normal-state-cursor '(nil))
        (setq-local evil-normal-state-cursor spray--last-evil-cursor-state)))
  (add-hook 'spray-mode-hook #'spray-mode-hide-cursor)
  (map! :map spray-mode-map
        "<return>" #'spray-start/stop
        "f" #'spray-faster
        "s" #'spray-slower
        "t" #'spray-time
        "<right>" #'spray-forward-word
        "h" #'spray-forward-word
        "<left>" #'spray-backward-word
        "l" #'spray-backward-word
        "q" #'spray-quit))
```

### 3.5.5 Elcord

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

```
(package! elcord :pin "64545671174f9ae307c0bd0aa9f1304d04236421")
```

```
(use-package! elcord
  :commands elcord-mode
  :config
  (setq elcord-use-major-mode-as-main-icon t))
```

## 3.6 File types

### 3.6.1 Authinfo

My patch giving my patch giving `authinfo-mode` syntax highlighting is only available in Emacs28+. For older versions, I've got a package I can use.

```
(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))
```

### 3.6.2 Systemd

For editing systemd unit files

```
(package! systemd :pin "b6ae63a236605b1c5e1069f7d3afe06ae32a7bae")
```

```
(use-package! systemd
:defer t)
```

### 3.6.3 Stan

Stan is probabilistic programming language written in C++. From my brief exposure I think of it as a nicer JAGS. Though Turing.jl looks nicer yet...

Anyway, the [stan-dev/stan-mode](#) repository contains a number of packages for working with Stan code. Let's grab them all.

```
(package! stan-mode :pin "9bb858b9f1314dcf1a5df23e39f9af522098276b")
(package! company-stan :pin "9bb858b9f1314dcf1a5df23e39f9af522098276b")
(package! eldoc-stan :pin "9bb858b9f1314dcf1a5df23e39f9af522098276b")
(package! flycheck-stan :pin "9bb858b9f1314dcf1a5df23e39f9af522098276b")
(package! stan-snippets :pin "9bb858b9f1314dcf1a5df23e39f9af522098276b")
```



# CHAPTER Applications 4

## 4.1 Ebooks



**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.

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 "cb93563d0ec9e0c653210bc574f9546d1e7db437")
```

Then for reading them, the only currently viable options seems to be [nov.el](#).

```
(package! nov :pin "b3c7cc28e95fe25ce7b443e5f49e2e45360944a3")
```

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

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

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

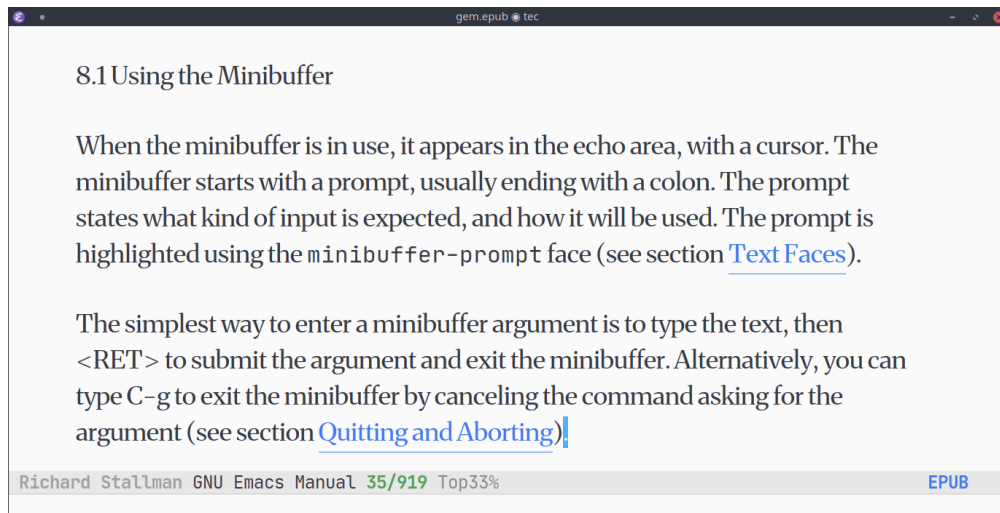
```

: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 81
  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))
    ,(propertyize
      "%P "
      'face 'doom-modeline-buffer-minor-mode)
    ,(propertyize
      " "
      '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))

```

## 4.2 Calculator

Emacs includes the venerable `calc`, which is a pretty impressive RPN (Reverse Polish Notation) calculator. However, we can do a bit to improve the experience.

### 4.2.1 Defaults

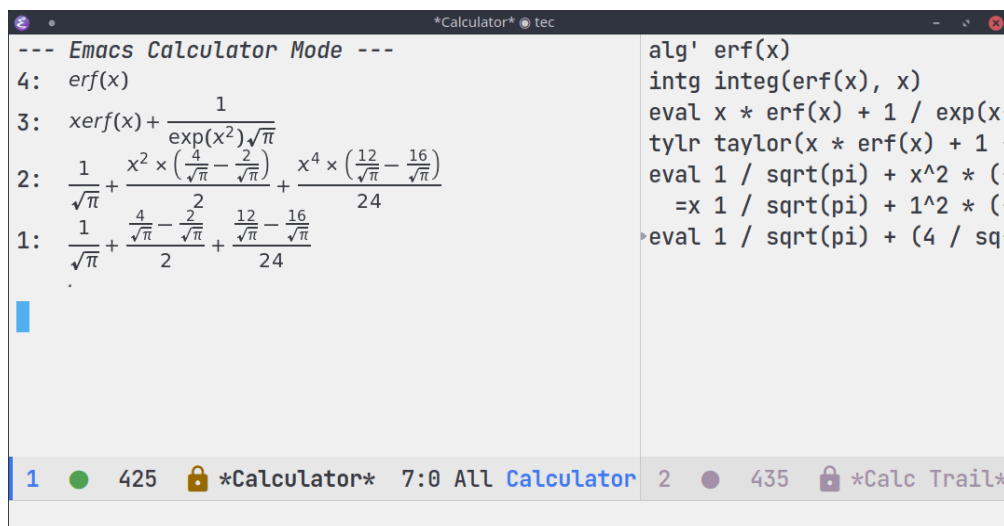
Any sane person prefers radians and exact values.

```
(setq calc-angle-mode 'rad ; radians are rad
      calc-symbolic-mode t) ; keeps expressions like \sqrt{2} irrational for as long
      ↪ as possible
```

### 4.2.2 CalcTeX

Everybody knows that mathematical expressions look best with  $\LaTeX$ , so `calc`'s ability to create  $\LaTeX$  representations of its expressions provides a lovely opportunity which is taken advantage of in the `CalcTeX` package.

```
(package! calcctx :recipe (:host github :repo "johnboughlin/calcctx"
                          :files ("*.el" "calcctx/*.el" "calcctx-contrib/*.el"
                                  ↪ "org-calcctx/*.el" "vendor"))
:pin "784cf911bc96aac0f47d529e8cee96ebd7cc31c9")
```



We'd like to use CalcTeX too, so let's set that up, and fix some glaring inadequacies — why on earth would you commit a hard-coded path to an executable that *only works on your local machine*, consequently breaking the package for everyone else!?

```
(use-package! calctex
  :commands calctex-mode
  :init
  (add-hook 'calc-mode-hook #'calctex-mode)
  :config
  (setq calctex-additional-latex-packages "
    \\usepackage[usenames]{xcolor}
    \\usepackage{soul}
    \\usepackage{adjustbox}
    \\usepackage{amsmath}
    \\usepackage{amssymb}
    \\usepackage{siunitx}
    \\usepackage{cancel}
    \\usepackage{mathtools}
    \\usepackage{mathalpha}
    \\usepackage{xparse}
    \\usepackage{arevmath}"
    calctex-additional-latex-macros
    (concat calctex-additional-latex-macros
      "\n\\let\\evalto\\Rightarrow"))
  (defadvice! no-messaging-a (orig-fn &rest args)
    :around #'calctex-default-dispatching-render-process
    (let ((inhibit-message t) message-log-max)
      (apply orig-fn args)))
  ;; Fix hardcoded dvichop path (whyyyyyyy)
  (let ((vendor-folder (concat (file-truename doom-local-dir)
    "straight/"
    (format "build-%s" emacs-version)
    "/calctex/vendor/"))))
    (setq calctex-dvichop-sty (concat vendor-folder "texd/dvichop")
          calctex-dvichop-bin (concat vendor-folder "texd/dvichop")))
  (unless (file-exists-p calctex-dvichop-bin)
    (message "CalcTeX: Building dvichop binary")
    (let ((default-directory (file-name-directory calctex-dvichop-bin)))
      (call-process "make" nil nil nil))))
```

### 4.2.3 Embedded calc

Embedded calc is a lovely feature which let's us use calc to operate on  $\text{\LaTeX}$  maths expressions. The standard keybinding is a bit janky however ( $\text{C-x} \star \text{e}$ ), so we'll add a localleader-based alternative.

```
(map! :map calc-mode-map
      :after calc
      :localleader
      :desc "Embedded calc (toggle)" "e" #'calc-embedded)
(map! :map org-mode-map
      :after org
      :localleader
      :desc "Embedded calc (toggle)" "E" #'calc-embedded)
(map! :map latex-mode-map
      :after latex
      :localleader
      :desc "Embedded calc (toggle)" "e" #'calc-embedded)
```

Unfortunately this operates without the (rather informative) calculator and trail buffers, but we can advise it that we would rather like those in a side panel.

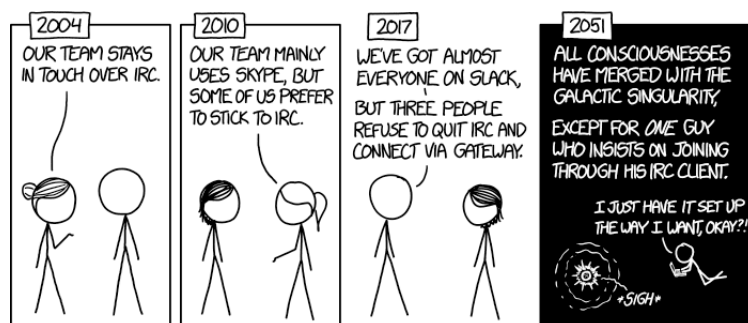
```
(defvar calc-embedded-trail-window nil)
(defvar calc-embedded-calculator-window nil)

(defadvice! calc-embedded-with-side-pannel (&rest _)
  :after #'calc-do-embedded
  (when calc-embedded-trail-window
    (ignore-errors
      (delete-window calc-embedded-trail-window))
    (setq calc-embedded-trail-window nil))
  (when calc-embedded-calculator-window
    (ignore-errors
      (delete-window calc-embedded-calculator-window))
    (setq calc-embedded-calculator-window nil))
  (when (and calc-embedded-info
             (> (* (window-width) (window-height)) 1200))
    (let ((main-window (selected-window))
          (vertical-p (> (window-width) 80)))
      (select-window
       (setq calc-embedded-trail-window
              (if vertical-p
                   (split-window-horizontally (- (max 30 (/ (window-width) 3))))
                   (split-window-vertically (- (max 8 (/ (window-height) 4)))))))
       (switch-to-buffer "*Calc Trail*")
       (select-window
        (setq calc-embedded-calculator-window
               (if vertical-p
                    (split-window-vertically -6)
                    (split-window-horizontally (- (/ (window-width) 2))))))
        (switch-to-buffer "*Calculator*")
        (select-window main-window))))
```

### 4.3 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 ~~reeluses~~ 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))

```

### 4.3.1 Org-style emphasis

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

```

(defun lui-org-to-irc ()
  "Examine a buffer with simple org-mode formatting, and converts the emphasis:
   *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))

```

### 4.3.2 Emojis

Let's setup Circe to use some emojis

```

(defun lui-ascii-to-emoji ()
  (goto-char (point-min))
  (while (re-search-forward "\\( \\)?:::\\([^\t: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" . "👅")
("zany" . "🤪")
("raised_eyebrow" . "🙄")
("monocle" . "👓")
("nerd" . "🤓")
("cool" . "😎")
("star_struck" . "🌟")
("party" . "🎉")
("smirk" . "😏")
("unamused" . "😒")
("disappointed" . "😞")
("pensive" . "😐")
("worried" . "😟")
("confused" . "😕")
("slight_frown" . "😏")
("frown" . "😞")
("persevere" . "😏")
("confounded" . "😏")
("tired" . "😫")
("weary" . "😫")
("pleading" . "😏")
("tear" . "😭")
("cry" . "😭")
("sob" . "😭")
("triumph" . "😏")
("angry" . "😡")
("rage" . "😡")
("exploding_head" . "💥")
("flushed" . "😳")
("hot" . "🔥")
("cold" . "❄️")
("scream" . "😱")
("fearful" . "😱")
("disappointed" . "😞")
("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" . "👍")
("thumbedown" . "👎")
("-1" . "👎")
("ok" . "👌")
("pinch" . "👉")
("left" . "👉")
("right" . "👈")
("down" . "👇")
("wave" . "👋")
("pray" . "🙏")
("eyes" . "👁")
("brain" . "🧠")
("facepalm" . "🤦")
("tada" . "👏")
("fire" . "🔥")
("flying_money" . "💸")
("lightbulb" . "💡")
("heart" . "💖")
("sparkling_heart" . "💖")
("heartbreak" . "💔")
("100" . "💯"))

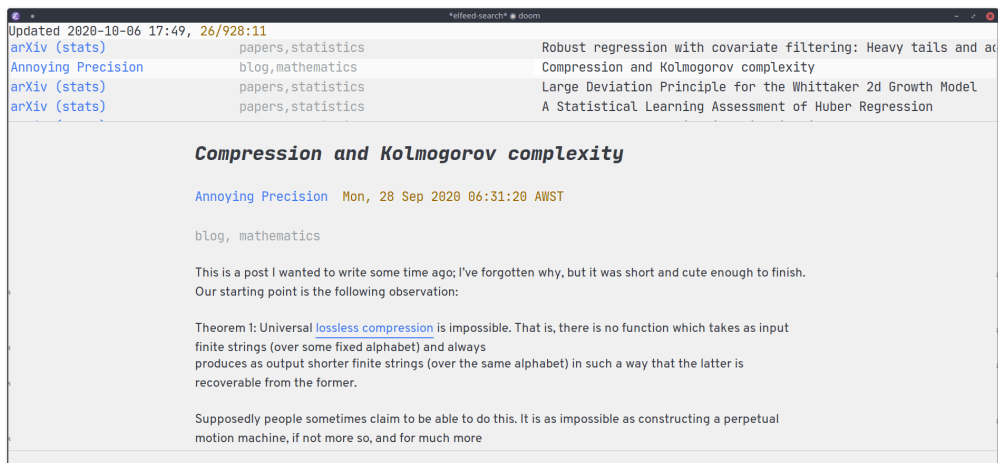
(defvar lui-emoticons-alist
  '((":" . "slight_smile")
    (";" . "wink")
    (":D" . "smile")
    ("=D" . "grin"))

```

```
( "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"))))
```

## 4.4 Newsfeed

RSS feeds are still a thing. Why not make use of them with `el feed`. 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.



### 4.4.1 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)
```

### 4.4.2 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))
```

### 4.4.3 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
  (setq-local truncate-lines nil
               shr-width 120
               visual-fill-column-center-text t
               default-text-properties '(line-height 1.1))
  (let ((inhibit-read-only t)
        (inhibit-modification-hooks 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)))
)

```

#### 4.4.4 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\\([^\r\n]+\\)" .
      ↪ "https://www.jstatsoft.org/index.php/jss/article/view/v0\\1/v\\1.pdf")
      ("http://arxiv.org/abs/\\([^\r\n]+\\)" . "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))))))

  )
```

## 4.5 Dictionary

Doom already loads `define-word`, and provides its 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 available, 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.

**Literate****Webster's Revised Unabridged Dictionary (1913)****Lit"er\*ate**, **adjective** [Latin *litteratus*, *litteratus*. See **Letter**.]

Instructed in learning, science, or literature; learned; lettered.

The literate now chose their emperor, as the military  
chose theirs. —*Landor*.

**Lit"er\*ate**, **noun**

1. One educated, but not having taken a university degree; especially, such a person who is prepared to take holy orders. [Eng.]
2. A literary man.

**Etymology****literate** **adjective**

"educated, instructed, having knowledge of letters," early 15c., from Latin *litteratus/litteratus* "educated, learned, who knows the letters;" formed in imitation of Greek *grammatikos* from Latin *littera/litera* "alphabetic letter" (see **letter** (noun 1)). By late 18c. especially "acquainted with literature." As a noun, "one who can read and write," 1894.

**Synonyms****adjective**

Learned, lettered.

```
(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 `sdv` to that in the future.

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

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))

```

Lastly, I want to make sure I have some dictionaries set up. I've put a tarball of dictionaries online which we can download if none seem to be present on the system.

```

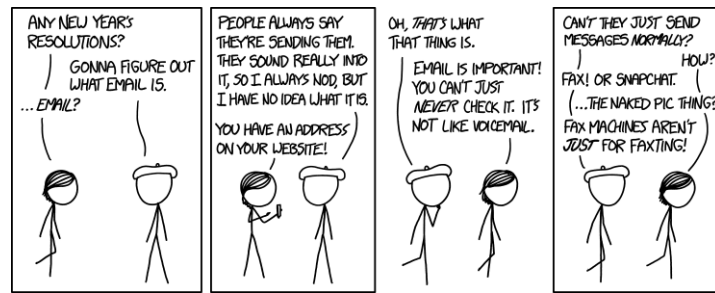
DIC_FOLDER=${STARDICT_DATA_DIR:-${XDG_DATA_HOME:-$HOME/.local/share}/stardict}/dic
if [ ! -d "$DIC_FOLDER" ]; then
  TMP=$(mktemp -d /tmp/dict-XXX)
  cd "$TMP"
  curl -A "Mozilla/4.0" -o "stardict.tar.gz"
  ↪ "https://tecosaur.com/resources/config/stardict.tar.gz"
  tar -xf "stardict.tar.gz"
  rm "stardict.tar.gz"
  mkdir -p "$DIC_FOLDER"
  mv * "$DIC_FOLDER"
fi

```

## 4.6 Mail

### 4.6.1 Fetching

The contenders for this seem to be:



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

- [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 not much 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 existence 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 existence")
  (defvar mu4e-reindex-request-min-seperation 5.0
    "Don't refresh again until this many second have elapsed.
     Prevents a series of redisplay 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 existence 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**

**onNewMailPost**



- 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 simultaneous 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
```

## 4.6.2 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 for 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 //'
```

### 4.6.3 Sending

[Smtplib](#) 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 `~/.config/msmtp/config`.

## 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/gsaslib/libgsasl-$GSASL_VERSION.tar.gz" | tar xz
curl "ftp://ftp.gnu.org/gnu/gsaslib/gsas-$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 msmtmp.

```
cd ~/.local/lib/
git clone https://github.com/marlam/msmtmp-mirror.git ./msmtmp
cd ./msmtmp
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/msmtmp "$@"
```

#### 4.6.4 Mu4e

Webmail clients are nice and all, but I still don't believe that SPAs in my browser can replace 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")
```

Alternatively, I may need to add the `/usr/share/` path.

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

Let's also just shove all the Elisp code here in an (`after! ...`) block.

```
(after! mu4e
  <mu4e-conf>>
)
```

#### 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

```
(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.

```
(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 a prompt otherwise also seems sensible.

We can register Emacs as a potential email client with a desktop file. We could put an `emacsclient` ...

entry in the Exec field, but I've found this a bit dodgy. Instead let's package the emacsclient behaviour in a little executable ~/.local/bin/emacsmail.

```
emacsclient -create-frame --alternate-editor='' --no-wait --eval \
"(progn (x-focus-frame nil) (mu4e-compose-from-mailto \"$1\" t))"
```

Now we can just call that in a desktop file.

```
[Desktop Entry]
Name=M4e
GenericName=Compose a new message with Mu4e in Emacs
Comment=Open mu4e compose window
MimeType=x-scheme-handler/mailto;
Exec=emacsmail %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 &optional quit-frame-after)
  (require 'mu4e)
  (unless mu4e~server-props (mu4e t) (sleep-for 0.1))
  (let* ((mailto (message-parse-mailto-url mailto-string))
        (to (cdr (assoc "To" mailto)))
        (subject (or (cdr (assoc "Subject" mailto)) ""))
        (body (cdr (assoc "Body" mailto)))
        (headers (-filter (lambda (spec) (not (-contains-p '("To" "Subject" "Body")
        ↪ (car spec)))) mailto)))
    (when-let ((mu4e-main (get-buffer mu4e-main-buffer-name)))
      (switch-to-buffer mu4e-main))
    (mu4e~compose-mail to subject headers)
    (when body
      (goto-char (point-min))
      (if (eq major-mode 'org-msg-edit-mode)
          (org-msg-goto-body)
          (mu4e-compose-goto-bottom))
      (insert body))
    (goto-char (point-min))
    (cond ((null to) (search-forward "To: "))
          ((string= "" subject) (search-forward "Subject: "))
          (t (if (eq major-mode 'org-msg-edit-mode)
                 (org-msg-goto-body)
```

```

        (mu4e-compose-goto-bottom)))
(font-lock-ensure)
(when evil-normal-state-minor-mode
  (evil-append 1))
(when quit-frame-after
  (add-hook 'kill-buffer-hook
    `((lambda ()
        (when (eq (selected-frame) ,(selected-frame))
          (delete-frame)))))))

```

It would also be nice to change the name pre-filled in From: when drafting.

```

(defvar mu4e-from-name "Timothy"
  "Name used in \"From:\" template.")
(defadvice! mu4e~draft-from-construct-renamed (orig-fn)
  "Wrap `mu4e~draft-from-construct-renamed' to change the name."
  :around #'mu4e~draft-from-construct
  (let ((user-full-name mu4e-from-name))
    (funcall orig-fn)))

```

We can also use this a signature,

```

(setq message-signature mu4e-from-name)

```

## Working with the Org mailing list

**Adding X-Woof headers** I'm fairly active on the Org mailing list (ML). The Org ML has a linked bug/patch tracker, <https://updates.orgmode.org/> managed by **Woof**. However, I feel like I spend too much time looking up what the appropriate headers are for updating the status of bugs and patches. What I need, is some sort of convenient tool. Let's write one.

First, a function that asks what I want to do and returns the appropriate X-Woof header.

```

(defun +mu4e-get-woof-header ()
  (pcase (read-char
    (format "\
    %s
    %s Declare %s Applied %s Aborted
    %s
    %s Confirm %s Fixed
    %s
    %s Request %s Resolved
    %s remove X-Woof header"
    (propertize "Patch" 'face 'outline-3)
    (propertize "p" 'face '(bold consult-key))

```

```

      (propertize "a" 'face '(bold consult-key))
      (propertize "c" 'face '(bold consult-key))
      (propertize "Bug" 'face 'outline-3)
      (propertize "b" 'face '(bold consult-key))
      (propertize "f" 'face '(bold consult-key))
      (propertize "Help" 'face 'outline-3)
      (propertize "h" 'face '(bold consult-key))
      (propertize "r" 'face '(bold consult-key))
      (propertize "x" 'face '(bold error)))
  (?p "X-Woof-Patch: confirmed")
  (?a "X-Woof-Patch: applied")
  (?c "X-Woof-Patch: cancelled")
  (?b "X-Woof-Bug: confirmed")
  (?f "X-Woof-Bug: fixed")
  (?h "X-Woof-Help: confirmed")
  (?r "X-Woof-Help: cancelled")
  (?x 'delete))

```

Now we just need a function which will add such a header to a buffer

```

(defun +mu4e-insert-woof-header ()
  "Insert an X-Woof header into the current message."
  (interactive)
  (when-let ((header (+mu4e-get-woof-header)))
    (save-excursion
      (goto-char (point-min))
      (search-forward "--text follows this line--")
      (unless (eq header 'delete)
        (beginning-of-line)
        (insert header "\n")
        (forward-line -1))
      (when (re-search-backward "^X-Woof-" nil t)
        (kill-whole-line)))))

(map! :map mu4e-compose-mode-map
      :localleader
      :desc "Insert X-Woof Header" "w" #' +mu4e-insert-woof-header)

(map! :map org-msg-edit-mode-map
      :after org-msg
      :localleader
      :desc "Insert X-Woof Header" "w" #' +mu4e-insert-woof-header)

```

Lovely! That should make adding these headers a breeze.

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



```

(after! mu4e
  (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)

  (define-minor-mode +org-ml-patchy-mood-mode
    "Apply patches to Org in bulk."
    :global t
    (let ((action (cons "apply patch to org" #'org-ml-apply-patch)))
      (if +org-ml-patchy-mood-mode
        (add-to-list 'mu4e-view-actions action)
        (setq mu4e-view-actions (delete action mu4e-view-actions)))))

  (defun +org-ml-apply-patch (msg)
    "Apply the patch in the current message to Org."
    (interactive)
    (unless msg (setq msg (mu4e-message-at-point)))
    (with-current-buffer (get-buffer-create "*Shell: Org apply patches*")
      (erase-buffer)
      (let* ((default-directory +org-ml-target-dir)
             (exit-code (call-process "git" nil t nil "am" (mu4e-message-field msg
               ↪ :path))))
        (magit-refresh)
        (when (not (= 0 exit-code))
          (+popup/buffer))))))

  (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")

```

```

(goto-char url-http-end-of-headers)
(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 (cdr (assoc (completing-read
                             "Thread: " (mapcar #'car patches))
                             patches))))
    (+workspace-switch +mu4e-workspace-name)
    (mu4e-view-message-with-message-id msg-id)
    (unless +org-ml-patchy-mood-mode
      (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)
  (+org-ml-apply-patch msg)))

```

**Mail list archive links** The other thing which it's good to be easily able to do is grab a link to the current message on <https://list.orgmode.org>.

```

(after! mu4e
  (defun +mu4e-ml-message-link (msg)
    "Copy the link to MSG on the mailing list archives."
    (let* ((list-addr (or (mu4e-message-field msg :mailing-list)
                          (cdar (mu4e-message-field-raw msg :list-post))
                          (thread-last (append (mu4e-message-field msg :to)
                                                (mu4e-message-field msg :cc))
                                         (mapcar #'last)
                                         (mapcar #'cdr)
                                         (mapcar (lambda (addr)
                                                    (when (string-match-p "emacs.*@gnu\\.org$" addr)
                                                      (replace-regexp-in-string "@\" \".\" addr)))))))

```

```

                                (delq nil)
                                (car)))
(msg-url
 (cond
  ((string= "emacs-orgmode.gnu.org" list-addr)
   (format "https://list.orgmode.org/%s" (mu4e-message-field msg
    ↪ :message-id)))
  (t (user-error "Mailing list %s not supported" list-addr))))
(message "Link %s copied to clipboard" (gui-select-text msg-url))
msg-url))

(add-to-list 'mu4e-view-actions (cons "link to message ML" #'mu4e-ml-message-link)
 ↪ t))

```

#### 4.6.5 OrgMsg

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@@html:<b>@@Timothy@@html:</b>@@\n#+end_signature")
(map! :map org-msg-edit-mode-map
      :after org-msg
      :n "G" #'org-msg-goto-body)

```

# CHAPTER 5

## Language configuration

### 5.1 General

#### 5.1.1 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)
```

### 5.2 Plaintext

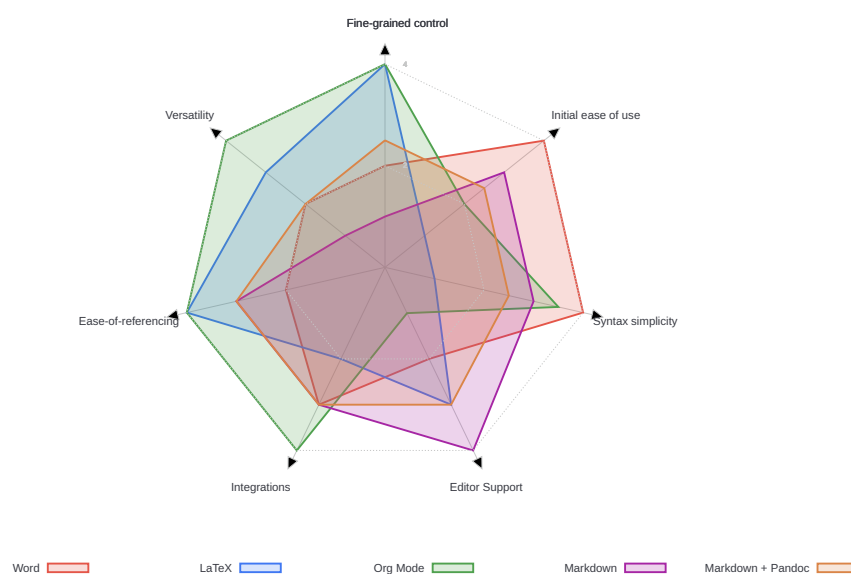
It's nice to see ANSI colour codes displayed. However, until Emacs 28 it's not possible to do this without modifying the buffer, so let's condition this block on that.

```
(after! text-mode
  (add-hook! 'text-mode-hook
    ;; Apply ANSI color codes
    (with-silent-modifications
      (ansi-color-apply-on-region (point-min) (point-max) t))))
```

### 5.3 Org

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

Format	Fine-grained control	Initial ease of use	Syntax simplicity	Editor Support	Integrations	Ease-of-referencing	Versatility
Word	2	4	4	2	3	2	2
LaTeX	4	1	1	3	2	4	3
Org Mode	4	2	3.5	1	4	4	4
Markdown	1	3	3	4	3	3	1
Markdown + Pandoc	2.5	2.5	2.5	3	3	3	2



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.

	pdfLaTeX	filter	style.scss	Weaving
	filter	filter	filter	(Documents)
	filter	filter	filter	
.tex	filter	filter	filter	
filter	filter	filter	filter	
graphc.png	filter	filter	filter	
image.jpeg	filter	filter	filter	
filter	filter	filter	filter	
figure.png	filter	filter	filter	
filter	filter	filter	filter	
filter	filter	filter	filter	
result	filter	filter	filter	
filter	filter	filter	filter	
execution	filter	filter	filter	
filter	filter	filter	filter	
code blocks	filter	filter	filter	
filter	filter	filter	filter	
	filter	filter	filter	
	filter	filter	filter	

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

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

### 5.3.1 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
```

## Git diffs

Protesilaos wrote a [very helpful article](#) in which he explains how to change the git diff chunk heading to something more useful than just the immediate line above the hunk — like the parent heading.

This can be achieved by first adding a new diff mode to git in `~/.config/git/attributes`

```
*.org diff=org
```

Then adding a regex for it to `~/.config/git/config`

```
[diff "org"]  
xfuncname = "^(\\"*+ +.*)" $"
```

## 5.3.2 Packages

### Org itself

There are actually three possible package statements I may want to use for Org.

If I'm on a machine where I can push changes, I want to be able to develop Org. I can check this by checking the content of the `SSH` key `~/.ssh/id_ed25519.pub`.

1. If this key exists and there isn't a repo at `$straight-base-dir/straight/repos/org-mode` with the right remote, we should install it as such.
2. If the key exists and repo are both set up, the package should just be ignored.
3. If the key does not exist, the Org's HEAD should just be used

To account for this situation properly, we need a short script to determine the correct package statement needed.

```
(load (expand-file-name "core/core.el" user-emacs-directory) nil t) ; for  
↪ `doom-local-dir'
```

```
(let ((dev-key (and (file-exists-p "~/.ssh/id_ed25519.pub")
                    (= 0 (shell-command "cat ~/.ssh/id_ed25519.pub | grep -q
                    ⇒ AAAAC3NzaC1lZDI1NTE5AAAAIOZZqcJ0LdN+QFHKyW8ST2zz750+8Tdv09IT5geXpQVt")))))
      (dev-pkg (let ((default-directory (expand-file-name "straight/repos/org-mode"
                  ⇒ doom-local-dir)))
                  (and (file-exists-p default-directory)
                       (string=
                        ⇒ "tec@git.savannah.gnu.org:/srv/git/emacs/org-mode.git\n"
                        ⇒ (shell-command-to-string "git remote get-url origin")))))
      (recipe-common '(:files ("*.el" "lisp/*.el" "etc")
                      :pre-build
                      (with-temp-file (doom-path (straight--repos-dir "org-mode")
                  ⇒ "org-version.el")
                        (insert "(fset 'org-release (lambda () \"9.5\")\n"
                                (format "(fset 'org-git-version (lambda ()
                  ⇒ \"%s\")\n"
                                (substring (shell-command-to-string "git
                  ⇒ rev-parse --short HEAD") 0 -1))
                                "(provide 'org-version)\n"))
                      :includes org)))
      (prin1-to-string
       `(package! org-mode
         :recipe (,@(cond ((and dev-key dev-pkg)
                           (list :host nil :repo nil :local-repo (expand-file-name
                  ⇒ "straight/repos/org-mode" doom-local-dir)))
                           (dev-key
                            (list :host nil :repo
                  ⇒ "tec@git.savannah.gnu.org:/srv/git/emacs/org-mode.git"))
                           (t
                            (list :host 'github :repo "emacs-straight/org-mode"))))
         ,@recipe-common)
         :pin nil)))
```

```
<<org-pkg-statement(>>
(unpin! org-mode) ; there be bugs
(package! org-contrib
 :recipe (:host nil :repo "https://git.sr.ht/~bzg/org-contrib"
         :files ("lisp/*.el"))
 :pin "b8012e759bd5bf5da802b0b41734a8fec218323c")
```

## Visuals

**Tables** 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! We'll make use of this with `writeroom-mode`.



```
(package! org-pretty-table
  :recipe (:host github :repo "Fuco1/org-pretty-table") :pin
  ↪ "87772a9469d91770f87bfa788580fca69b9e697a")
```

```
(use-package! org-pretty-table
  :commands (org-pretty-table-mode global-org-pretty-table-mode))
```

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

```
(package! org-appear :recipe (:host github :repo "awth13/org-appear")
  :pin "148aa124901ae598f69320e3dcada6325cdc2cf0")
```

```
(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-elements'
  ;; needs to be run after other hooks have acted.
  (run-at-time nil nil #'org-appear--set-elements))
```

**Heading structure** Speaking of headlines, a nice package for viewing and managing the heading structure has come to my attention.

```
(package! org-ol-tree :recipe (:host github :repo "Townk/org-ol-tree")
  :pin "207c748aa5fea8626be619e8c55bdb1c16118c25")
```

```
(use-package! org-ol-tree
  :commands org-ol-tree)
(map! :map org-mode-map
  :after org
  :localleader
  :desc "Outline" "O" #'org-ol-tree)
```

## Extra functionality

**Org ref** Now and then citations need to happen

```
(package! org-ref :pin "3ca9beb744621f007d932deb8a4197467012c23a")
```

**Julia support** `ob-julia` is currently a bit borked, but there's an effort to improve this.

```
(package! ob-julia :recipe (:local-repo "lisp/ob-julia" :files ("*.el" "julia")))

(use-package! ob-julia
  :commands org-babel-execute:julia
  :config
  (setq org-babel-julia-command-arguments
    `("--sysimage"
      ,(when-let ((img "~/local/lib/julia.so")
        (exists? (file-exists-p img)))
        (expand-file-name img))
      "--threads"
      ,(number-to-string (- (doom-system-cpus) 2))
      "--banner=no")))
```

**HTTP requests** I like the idea of being able to make HTTP requests with Babel.

```
(package! ob-http :pin "b1428ea2a63bcb510e7382a1bf5fe82b19c104a7")

(use-package! ob-http
  :commands org-babel-execute:http)
```

**Transclusion** There's a really cool package in development to *transclude* Org document content.

```
(package! org-transclusion :recipe (:host github :repo "nobirot/org-transclusion")
  :pin "8cbbade1e3237200c2140741f39ff60176e703e7")

(use-package! org-transclusion
  :commands org-transclusion-mode
  :init
  (map! :after org :map org-mode-map
    "<f12>" #'org-transclusion-mode))
```

**Heading graph** Came across this and ... it's cool

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

**Cooking recipes** I need this in my life. It takes 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 "a97232b4706869ecae16a1352487a99bc3cf97af")
```

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))
```

**Importing with Pandoc** 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)
```

**Document comparison** It's quite nice to compare Org files, and the richest way to compare content is probably `latexdiff`. There are a few annoying steps involved here, and so I've written a package to streamline the process.

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

The only little annoyance is the fact that `latexdiff` uses `#FF0000` and `#0000FF` as the red/blue change indication colours. We can make this a bit nicer by post-processing the `latexdiff` result.

```
(use-package! orgdiff
  :defer t
  :config
  (defun +orgdiff-nicer-change-colours ()
    (goto-char (point-min))
    ;; Set red/blue based on whether chameleon is being used
    (if (search-forward "% make document follow Emacs theme" nil t)
        (setq red (substring (doom-blend 'red 'fg 0.8) 1)
              blue (substring (doom-blend 'blue 'teal 0.6) 1))
        (setq red "c82829"
              blue "00618a"))
    (when (and (search-forward "%DIF PREAMBLE EXTENSION ADDED BY LATEXDIFF" nil t)
               (search-forward "\\RequirePackage{color}" nil t))
      (when (re-search-forward "definecolor{red}{rgb}{1,0,0}" (cdr
        ↪ (bounds-of-thing-at-point 'line)) t)
        (replace-match (format "definecolor{red}{HTML}{%s}" red)))
      (when (re-search-forward "definecolor{blue}{rgb}{0,0,1}" (cdr
        ↪ (bounds-of-thing-at-point 'line)) t)
        (replace-match (format "definecolor{blue}{HTML}{%s}" blue))))))
```

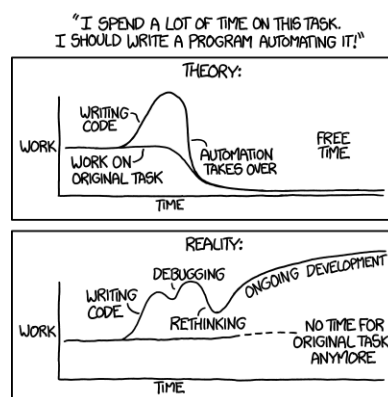
```
(add-to-list 'orgdiff-latexdiff-postprocess-hooks #'orgdiff-nicer-change-colours))
```

**Org music** It's nice to be able to link to music

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

```
(use-package! org-music
  :after org
  :config
  (setq org-music-mpris-player "Lollypop"
        org-music-track-search-method 'beets
        org-music-beets-db "~/Music/library.db"))
```

### 5.3.3 Behaviour



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

### 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 convenient
      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 accidentally do weird
      ⇨ stuff in invisible regions)
```

```
org-export-with-sub-superscripts '{})      ; don't treat lone _ / ^ as
↳ sub/superscripts, require _{} / ^{}
```

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,  $\text{\LaTeX}$ ) 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))
```

**The utility of zero-width spaces** Occasionally in Org you run into annoyances where you want to have two separate blocks right together without a space. For example, to **emphasise** part of a word, or put a currency symbol immediately before an inline source block. There is a solution to this, it just sounds slightly hacky — zero width spaces. Because this is Emacs, we can make this feel much less hacky by making a minor addition to the Org key map ȷ.

```
(map! :map org-mode-map
      :nie "M-SPC M-SPC" (cmd! (insert "\u200B")))
```

We then want to stop the space from being included in exports, which can be done with a little filter.

```
(defun +org-export-remove-zero-width-space (text _backend _info)
  "Remove zero width spaces from TEXT."
  (unless (org-export-derived-backend-p 'org)
    (replace-regexp-in-string "\u200B" "" text)))

(after! ox
  (add-to-list 'org-export-filter-final-output-functions
    ↪ #' +org-export-remove-zero-width-space t))
```

**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, and `org-ref` is *the* package for that.

Unfortunately, it ignores the `file = {...}.bib` keys though. Let's fix that. I separate files on ;, which may just be a Zotero/BetterBibLaTeX thing, but it's a good idea in my case at least.

```
(use-package! org-ref
  ;; :after org
  :defer t
  :config
  (defadvice! org-ref-open-bibtex-pdf-a ()
    :override #'org-ref-open-bibtex-pdf
    (save-excursion
      (bibtex-beginning-of-entry)
```

```

(let* ((bibtex-expand-strings t)
      (entry (bibtex-parse-entry t))
      (key (reftex-get-bib-field "=key=" entry))
      (pdf (or
            (car (-filter (lambda (f) (string-match-p "\\..pdf$" f))
                          (split-string (reftex-get-bib-field "file" entry)
                                         ↪ ";"))
            (funcall org-ref-get-pdf-filename-function key))))
      (if (file-exists-p pdf)
          (org-open-file pdf)
          (ding))))
(defadvice! org-ref-open-pdf-at-point-a ()
  "Open the pdf for bibtex key under point if it exists."
  :override #'org-ref-open-pdf-at-point
  (interactive)
  (let* ((results (org-ref-get-bibtex-key-and-file))
        (key (car results))
        (pdf-file (funcall org-ref-get-pdf-filename-function key)))
    (with-current-buffer (find-file-noselect (cdr results))
      (save-excursion
        (bibtex-search-entry (car results))
        (org-ref-open-bibtex-pdf))))))

```

There's also the new `org-cite` though. It would be nice to try that out.

To improve `org-cite`.

```

(package! citar :pin "a6926650114a8091f98bff8c7fd00add82043190")
(package! citeproc :pin "38e70c0a94eeefe86ddefc38dfa8ab2311008774")
(package! org-cite-csl-activate :recipe (:host github :repo
↪ "andras-simonyi/org-cite-csl-activate") :pin
↪ "8f49ccbd337edda01e52da0c75f6a76e2cc976f7")

```

```

(use-package! citar
  :when (featurep! :completion vertico))

(use-package! citeproc
  :defer t)

;;; Org-Cite configuration

(map! :after org
      :map org-mode-map
      :localleader
      :desc "Insert citation" "@ " #'org-cite-insert)

(use-package! oc
  :after org citar

```

```

:config
(require 'ox)
(setq org-cite-global-bibliography
  (let ((paths (or citar-bibliography
                  (bound-and-true-p bibtex-completion-bibliography))))
    ;; Always return bibliography paths as list for org-cite.
    (if (stringp paths) (list paths) paths)))
;; setup export processor; default csl/citeproc-el, with biblatex for latex
(setq org-cite-export-processors
  '((t csl)))

;;; Org-cite processors
(use-package! oc-biblatex
  :after oc)

(use-package! oc-csl
  :after oc
  :config
  (setq org-cite-csl-styles-dir "~/Zotero/styles"))

(use-package! oc-natbib
  :after oc)

;;; Third-party

(use-package! citar-org
  :no-require
  :custom
  (org-cite-insert-processor 'citar)
  (org-cite-follow-processor 'citar)
  (org-cite-activate-processor 'citar)
  (org-support-shift-select t)
  (when (featurep! :lang org +roam2)
    ;; Include property drawer metadata for 'org-roam' v2.
    (citar-org-note-include '(org-id org-roam-ref)))
  ;; Personal extras
  (setq citar-symbols
    `((file ,(all-the-icons-faicon "file-o" :v-adjust -0.1) . " ")
      (note ,(all-the-icons-material "speaker_notes" :face 'all-the-icons-silver
        ⇨ :v-adjust -0.3) . " ")
      (link ,(all-the-icons-octicon "link" :face 'all-the-icons-dsilver :v-adjust
        ⇨ 0.01) . " "))))

(use-package! oc-csl-activate
  :after oc
  :config
  (setq org-cite-csl-activate-use-document-style t)
  (defun +org-cite-csl-activate/enable ()

```



```
(interactive)
(setq org-cite-activate-processor 'csl-activate)
(add-hook! 'org-mode-hook '((lambda () (cursor-sensor-mode 1))
  ⇒ org-cite-csl-activate-render-all))
(defadvice! +org-cite-csl-activate-render-all-silent (orig-fn)
  :around #'org-cite-csl-activate-render-all
  (with-silent-modifications (funcall orig-fn)))
(when (eq major-mode 'org-mode)
  (with-silent-modifications
    (save-excursion
      (goto-char (point-min))
      (org-cite-activate (point-max)))
    (org-cite-csl-activate-render-all)))
(fmakunbound #'org-cite-csl-activate/enable)))
```

I think it would be nice to have a function to convert org-ref citations to org-cite

```
(after! oc
  (defun org-ref-to-org-cite ()
    "Attempt to convert org-ref citations to org-cite syntax."
    (interactive)
    (let* ((cite-conversions '(("cite" . "//b") ("Cite" . "//bc")
      ("nocite" . "/n")
      ("citep" . "") ("citep*" . "//f")
      ("parencite" . "") ("Parencite" . "//c")
      ("citeauthor" . "/a/f") ("citeauthor*" . "/a")
      ("citeyear" . "/na/b")
      ("Citep" . "//c") ("Citealp" . "//bc")
      ("Citeauthor" . "/a/cf") ("Citeauthor*" . "/a/c")
      ("autocite" . "") ("Autocite" . "//c")
      ("notecite" . "/l/b") ("Notecite" . "/l/bc")
      ("pnotecite" . "/l") ("Pnotecite" . "/l/bc"))))
      (cite-regexp (rx (regexp (regexp-opt (mapcar #'car cite-conversions) t))
        ":" (group (+ (not (any "\n" ,.))}))))))
    (save-excursion
      (goto-char (point-min))
      (while (re-search-forward cite-regexp nil t)
        (message (format "[cite%s:@%s]"
          (cdr (assoc (match-string 1) cite-conversions))
          (match-string 2)))
          (replace-match (format "[cite%s:@%s]"
            (cdr (assoc (match-string 1) cite-conversions))
            (match-string 2)))))))
```

**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 block."
  (setq centaur-lsp 'lsp-mode)
  (cl-check-type lang stringp)
  (let* ((edit-pre (intern (format "org-babel-edit-pre:%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** 'localleader v has no pre-existing binding, so I may as well use it with the same functionality as in  $\text{\LaTeX}$ . 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

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

```
(package! org-super-agenda :pin "a5557ea4f51571ee9def3cd9a1ab1c38f1a27af7")
```

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

```
(after! org-agenda
  (org-super-agenda-mode))
```

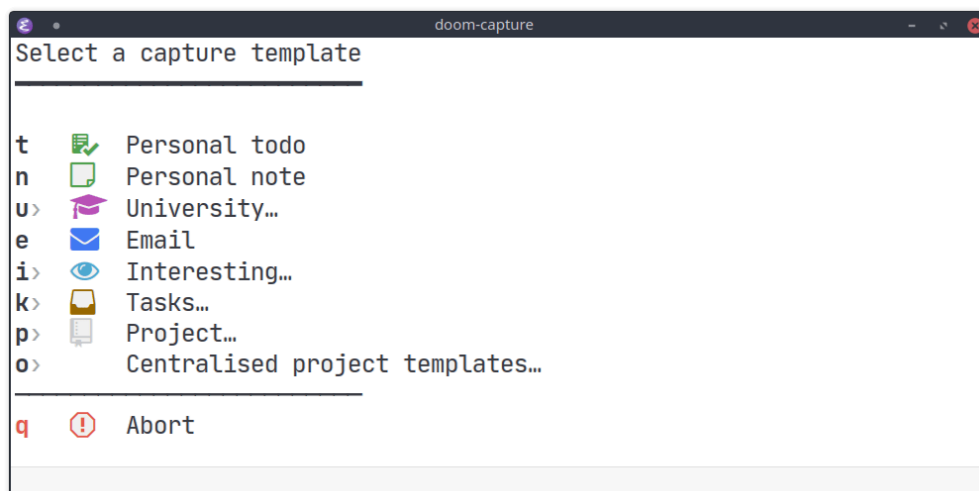
```
(setq org-agenda-skip-scheduled-if-done t
      org-agenda-skip-deadline-if-done t
      org-agenda-include-deadlines t
      org-agenda-block-separator nil
      org-agenda-tags-column 100 ;; from testing this seems to be a good value
      org-agenda-compact-blocks t)

(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.



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 "67fc46c8a68989b932bce879fbaa62c6a2456a1f")
```

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

```
(after! org-capture
  <<prettify-capture>>

  (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))

  (defvar +org-capture-recipes "~/Desktop/TEC/Organisation/recipes.org")

  (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"))
("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-recipes
: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\niiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiiii"
          "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-prettier (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
      (not (pos-visible-in-window-p (1-
        ⇒ (point-max)))))))
      (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)
```

**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)))
```

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

```
(package! org-roam-ui :recipe (:host github :repo "org-roam/org-roam-ui" :files
  ↳ ("*.el" "out")) :pin "cd1aefd56f648d32a25aae672ac1ab90893c0133")
(package! websocket :pin "fda445533309545c0787a79d73c19ddbeb57980") ; dependency of
  ↳ `org-roam-ui'
```

```
(use-package! websocket
  :after org-roam)

(use-package! org-roam-ui
  :after org-roam
```

```
:commands org-roam-ui-open
:hook (org-roam . org-roam-ui-mode)
:config
(require 'org-roam) ; in case autoloaded
(defun org-roam-ui-open ()
  "Ensure the server is active, then open the roam graph."
  (interactive)
  (unless org-roam-ui-mode (org-roam-ui-mode 1))
  (browse-url-xdg-open (format "http://localhost:%d" org-roam-ui-port))))
```

## Nicer generated heading IDs

Thanks to alphapapa's [unpackaged.el](#).

By default, Org generated heading IDs like #org80fc2a5 which ... works, but has two issues

- It's completely uninformative, I have no idea what's being referenced
- If I export the same file, everything will change. Now, while without hardcoded values it's impossible to set references in stone, it would be nice for there to be a decent chance of staying the same.

Both of these issues can be addressed by generating IDs like #language-configuration, which is what I'll do here.

It's worth noting that alphapapa's use of `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 characters 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 feasible for this particular situation, rather than the universal maximum average. This budget-adjusted per-word maximum length is given by the mathematical expression below:

$$\text{max length} = \lfloor \frac{\text{total length} - \text{chars for separators} - \sum_{\text{word}} \text{length(word)} \cdot \text{num(words)} > \text{average length}}{\text{average length}} \rfloor$$

↪ truncate 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
              ↪ separator-type chars with space
                        puny-encode-string
                        (replace-regexp-in-string "^xn--\\([.*?\\|) ?-?[\\([a-z0-9]+\\)"
              ↪ "\\2 \\1") ; rearrange punycode
                        (replace-regexp-in-string "[^A-Za-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 #'unpacked/org-export-get-reference)))
(unpacked/org-export-html-with-useful-ids-mode 1) ; ensure enabled, and advice run

(defun unpacked/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
                      (unpacked/org-export-new-named-reference datum cache))
                    (when (member (car datum) '(src-block table example
                    ↪ fixed-width property-drawer))
                      ;; Nameable elements
                      (unpacked/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 #'unpacked/org-export-html-with-useful-ids-mode)

```

We also need to redefine (`org-export-format-reference`) as it now may be passed a string as well as a number.

```

(defadvice! org-export-format-reference-a (reference)
  "Format REFERENCE into a string.
  REFERENCE is either a number or a string representing a reference,
  as returned by `org-export-new-reference'."
  :override #'org-export-format-reference
  (if (stringp reference) reference (format "org%07x" reference)))

```

## Nicer `org-return`

Once again, from [unpacked.el](#)

```

(defun unpacked/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))
        (unpacked/org-element-descendant-of type parent))))

;;;###autoload
(defun unpacked/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))
               (while (not (looking-back "\\(?:[:blank:]]?\\n\\|\\{3\\}" nil))
                 (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 Helpers

I often want to set `src-block` headers, and it's a pain to

- type them out
- remember what the accepted values are

- oh, and specifying the same language again and again

We can solve this in three steps

- having one-letter snippets, conditioned on (point) being within a src header
- creating a nice prompt showing accepted values and the current default
- pre-filling the src-block language with the last language used

For header args, the keys I'll use are

- r for :results
- e for :exports
- v for :eval
- s for :session
- d for :dir

```
(defun +yas/org-src-header-p ()
  "Determine whether `point' is within a src-block header or header-args."
  (pcase (org-element-type (org-element-context))
    ('src-block (< (point) ; before code part of the src-block
                  (save-excursion (goto-char (org-element-property :begin
                                                                    ↪ (org-element-context)))
                                (forward-line 1)
                                (point))))
    ('inline-src-block (< (point) ; before code part of the inline-src-block
                          (save-excursion (goto-char (org-element-property :begin
                                                                    ↪ (org-element-context)))
                                (search-forward "]{")
                                (point))))
    ('keyword (string-match-p "^header-args" (org-element-property :value
                                                                    ↪ (org-element-context))))))
```

Now let's write a function we can reference in yasnippets to produce a nice interactive way to specify header args.

```
(defun +yas/org-prompt-header-arg (arg question values)
  "Prompt the user to set ARG header property to one of VALUES with QUESTION.
  The default value is identified and indicated. If either default is selected,
  or no selection is made: nil is returned."
  (let* ((src-block-p (not (looking-back "^#\\++property:[ \\t]+header-args:.*"
                                         ↪ (line-beginning-position))))
    (default
     (or
      (cdr (assoc arg
                  (if src-block-p
```

```

        (nth 2 (org-babel-get-src-block-info t))
      (org-babel-merge-params
       org-babel-default-header-args
       (let ((lang-headers
              (intern (concat "org-babel-default-header-args:"
                              (+yas/org-src-lang)))))
         (when (boundp lang-headers) (eval lang-headers t))))))
    ""))
  default-value)
(setq values (mapcar
  (lambda (value)
    (if (string-match-p (regexp-quote value) default)
        (setq default-value
              (concat value " "
                    (propertize "(default)" 'face
                              ↪ 'font-lock-doc-face)))
        value))
  values))
(let ((selection (consult--read values :prompt question :default default-value)))
  (unless (or (string-match-p "(default)$" selection)
              (string= "" selection))
    selection)))

```

Finally, we fetch the language information for new source blocks.

Since we're getting this info, we might as well go a step further and also provide the ability to determine the most popular language in the buffer that doesn't have any header-args set for it (with #+properties).

```

(defun +yas/org-src-lang ()
  "Try to find the current language of the src/header at `point'.
  Return nil otherwise."
  (let ((context (org-element-context)))
    (pcase (org-element-type context)
      ('src-block (org-element-property :language context))
      ('inline-src-block (org-element-property :language context))
      ('keyword (when (string-match "^header-args:\\\([^\ ]+\)" (org-element-property
        ↪ :value context))
                    (match-string 1 (org-element-property :value context))))))

(defun +yas/org-last-src-lang ()
  "Return the language of the last src-block, if it exists."
  (save-excursion
    (beginning-of-line)
    (when (re-search-backward "[\t]*#\|+begin_src" nil t)
      (org-element-property :language (org-element-context))))

(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 (re-search-forward "[ \t]*#\\+begin_src" nil t)
      (push (+yas/org-src-lang) src-langs))
    (goto-char (point-min))
    (while (re-search-forward "[ \t]*#\\+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 edge cases, but should mostly work.

```

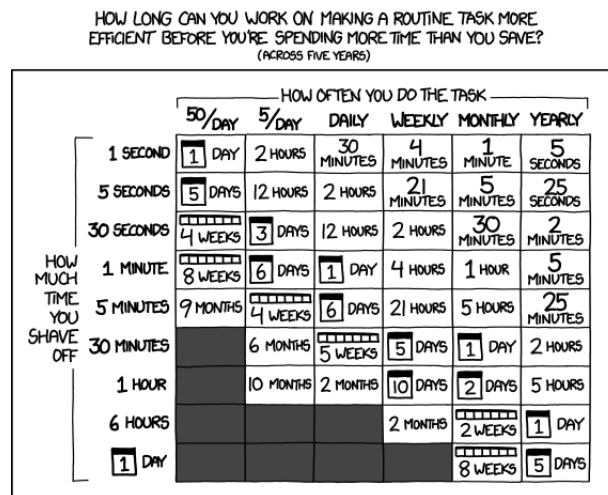
(defun org-syntax-convert-keyword-case-to-lower ()
  "Convert all #+KEYWORDS to #+keywords."
  (interactive)
  (save-excursion
    (goto-char (point-min))
    (let ((count 0))
      (case-fold-search nil)
      (while (re-search-forward "[ \t]*#\\+[A-Z_]+" nil t)
        (unless (s-matches-p "RESULTS" (match-string 0))
          (replace-match (downcase (match-string 0)) t)
          (incf count)))))

```

```
(setq count (1+ count)))
(message "Replaced %d occurrences" 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-fn #' +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 "\\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

**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))))
```

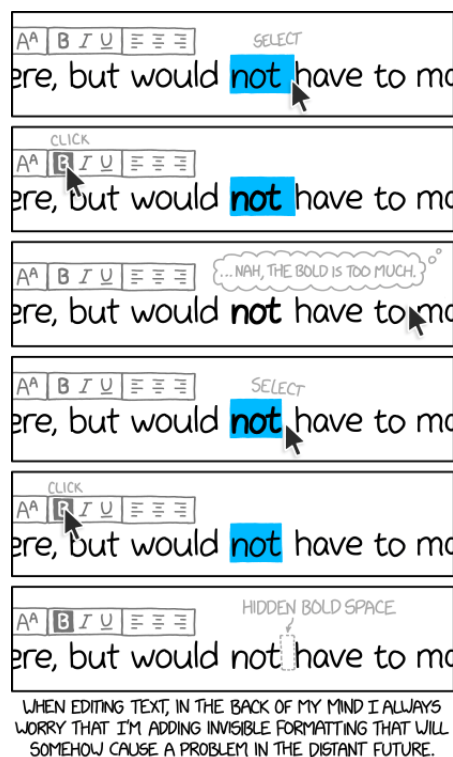
## Fix problematic hooks

When one of the `org-mode-hook` functions errors, it halts the hook execution. This is problematic, and there are two hooks in particular which cause issues. Let's make their failure less eventful.

```
(defadvice! shut-up-org-problematic-hooks (orig-fn &rest args)
  :around #'org-fancy-priorities-mode
  :around #'org-superstar-mode
  (ignore-errors (apply orig-fn args)))
```

## Flycheck with org-lint

Org may be simple, but that doesn't mean there's no such thing as malformed Org. Thankfully, malformed Org is a much less annoying affair than malformed zipped XML (looks at DOCX/ODT...), particularly because there's a rather helpful little tool called `org-lint` bundled with Org that can tell you about your mistakes.



**Invisible Formatting** To avoid errors like this, we render all text and pipe it through OCR before processing, fixing a handful of irregular bugs by burying them beneath a smooth, uniform layer of bugs.

Flycheck doesn't currently support Org, and there's aren't any packages to do so `;`. However, in an

issue on `org-lint` there is [some code](#) which apparently works. Surely this is what the clipboard was invented for? With that said, let's regurgitate the code, cross our fingers, and hope it works.

```
(defconst flycheck-org-lint-form
  (flycheck-prepare-emacs-lisp-form
   (require 'org)
   (require 'org-attach)
   (let ((source (car command-line-args-left))
         (process-default-directory default-directory))
     (with-temp-buffer
      (insert-file-contents source 'visit)
      (setq buffer-file-name source)
      (setq default-directory process-default-directory)
      (delay-mode-hooks (org-mode))
      (setq delayed-mode-hooks nil)
      (dolist (err (org-lint))
        (let ((inf (cl-second err)))
          (princ (elt inf 0))
          (princ ": ")
          (princ (elt inf 2))
          (terpri)))))))

(defconst flycheck-org-lint-variables
  '(org-directory
    org-id-locations
    org-id-locations-file
    org-attach-id-dir
    org-attach-use-inheritance
    org-attach-id-to-path-function-list
    org-link-parameters)
  "Variables inherited by the org-lint subprocess.")

(defun flycheck-org-lint-variables-form ()
  (require 'org-attach) ; Needed to make variables available
  `(progn
    ,@(seq-map (lambda (opt) `(setq-default ,opt ',(symbol-value opt)))
              (seq-filter #'boundp flycheck-org-lint-variables)))

(flycheck-define-checker org-lint
  "Org buffer checker using `org-lint'."
  :command ("emacs" (eval flycheck-emacs-args)
            "--eval" (eval (concat "(add-to-list 'load-path \""
                                   (file-name-directory (locate-library "org"))
                                   "\""))
            "--eval" (eval (flycheck-sexp-to-string
                           (flycheck-org-lint-variables-form)))
            "--eval" (eval (flycheck-sexp-to-string
                           (flycheck-org-lint-customisations-form)))
```

```

    "--eval" (eval flycheck-org-lint-form)
    "--" source)
:error-patterns
((error line-start line ": " (message) line-end))
:modes org-mode)

```

Turns out it almost works. Running `M-x flycheck-verify-setup` after running that snippet produces the following:

```

The following syntax checkers are not registered:
- org-lint
Try adding these syntax checkers to `flycheck-checkers'.

```

Well that's very nice and helpful. We'll just do that `;`.

```
(add-to-list 'flycheck-checkers 'org-lint)
```

It was missing custom link types, but that's easily fixed just by adding `org-link-parameters` to `flycheck-org-lint-variables`.

One remaining little annoyance is that it reports extra `#+options` that I've added to Org as errors. So we need to tell `org-lint` about them without having it load my whole config. Code duplication isn't great, but at least this isn't much.

```

(defun flycheck-org-lint-customisations-form ()
  `(progn
    (require 'ox)
    (cl-pushnew '(:latex-cover-page nil "coverpage" nil)
      (org-export-backend-options (org-export-get-backend 'latex)))
    (cl-pushnew '(:latex-font-set nil "fontset" nil)
      (org-export-backend-options (org-export-get-backend 'latex)))))

```

### 5.3.4 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.

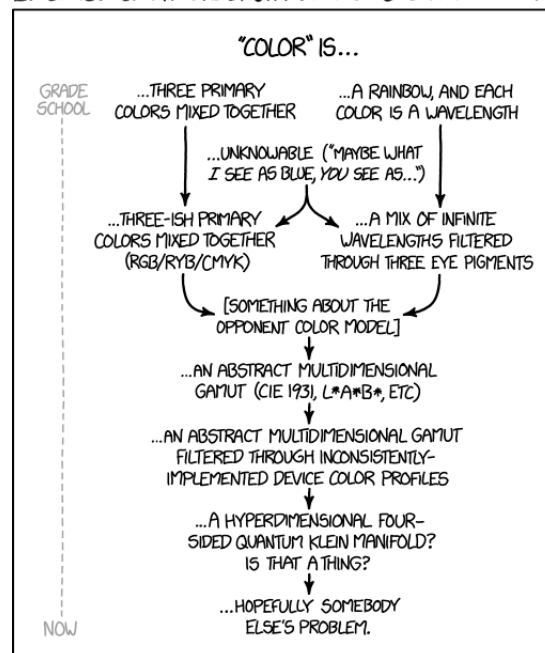
#### Font Display

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

```
(add-hook 'org-mode-hook #'org-pretty-mode)
```

Let's make headings a bit bigger

## EVOLUTION OF MY UNDERSTANDING OF COLOR OVER TIME:



**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.

```

(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)
```

Org files can be rather nice to look at, particularly with some of the customisations here. This comes at a cost however, expensive font-lock. Feeling like you're typing through molasses in large files is no fun, but there is a way I can defer font-locking when typing to make the experience more responsive.

```
(defun locally-defer-font-lock ()
  "Set jit-lock defer and stealth, when buffer is over a certain size."
  (when (> (buffer-size) 50000)
    (setq-local jit-lock-defer-time 0.05
                jit-lock-stealth-time 1)))

(add-hook 'org-mode-hook #'locally-defer-font-lock)
```

Apparently this causes issues with some people, but I haven't noticed anything problematic beyond the expected slight delay in some fontification, so until I do I'll use the above.

### Fontifying inline src blocks

Org does lovely things with `#+begin_src` blocks, like using font-lock for language's major-mode behind the scenes and pulling out the lovely colourful results. By contrast, inline `src_` blocks are somewhat neglected.

I am not the first person to feel this way, thankfully others have [taken to stackexchange](#) to voice their desire for inline src fontification. I was going to steal their work, but unfortunately they didn't perform *true* source code fontification, but simply applied the `org-code` face to the content.

We can do better than that, and we shall! Using `org-src-font-lock-fontify-block` we can apply language-appropriate syntax highlighting. Then, continuing on to `{{{results(...)}}}`, it can have the `org-block` face applied to match, and then the value-surrounding constructs hidden by mimicking the behaviour of `prettyfy-symbols-mode`.

### ❖ Warning

This currently only highlights a single inline src block per line. I have no idea why it stops, but I'd rather it didn't. If you have any idea what's going on or how to fix this *please* get in touch.

```
(defvar org-prettyfy-inline-results t
  "Whether to use (ab)use prettyfy-symbols-mode on {{{results(...)}}}.
   Either t or a cons cell of strings which are used as substitutions
   for the start and end of inline results, respectively.")
```

```

(defvar org-fontify-inline-src-blocks-max-length 200
  "Maximum content length of an inline src block that will be fontified.")

(defun org-fontify-inline-src-blocks (limit)
  "Try to apply `org-fontify-inline-src-blocks-1'."
  (condition-case nil
    (org-fontify-inline-src-blocks-1 limit)
    (error (message "Org mode fontification error in %S at %d"
                    (current-buffer)
                    (line-number-at-pos))))))

(defun org-fontify-inline-src-blocks-1 (limit)
  "Fontify inline src_LANG blocks, from `point' up to LIMIT."
  (let ((case-fold-search t)
        (initial-point (point)))
    (while (re-search-forward "\\_<src_\\([^\t\n[{}+\\)]{0,?}" limit t) ; stolen from
            ↪ `org-element-inline-src-block-parser'
            (let ((beg (match-beginning 0))
                  pt
                  (lang-beg (match-beginning 1))
                  (lang-end (match-end 1)))
              (remove-text-properties beg lang-end '(face nil))
              (font-lock-append-text-property lang-beg lang-end 'face 'org-meta-line)
              (font-lock-append-text-property beg lang-beg 'face 'shadow)
              (font-lock-append-text-property beg lang-end 'face 'org-block)
              (setq pt (goto-char lang-end))
              ;; `org-element--parse-paired-brackets' doesn't take a limit, so to
              ;; prevent it searching the entire rest of the buffer we temporarily
              ;; narrow the active region.
              (save-restriction
                (narrow-to-region beg (min (point-max) limit (+ lang-end
                ↪ org-fontify-inline-src-blocks-max-length)))
                (when (ignore-errors (org-element--parse-paired-brackets ?\{))
                  (remove-text-properties pt (point) '(face nil))
                  (font-lock-append-text-property pt (point) 'face 'org-block)
                  (setq pt (point)))
                (when (ignore-errors (org-element--parse-paired-brackets ?\{))
                  (remove-text-properties pt (point) '(face nil))
                  (font-lock-append-text-property pt (1+ pt) 'face '(org-block shadow))
                  (unless (= (1+ pt) (1- (point)))
                    (if org-src-fontify-natively
                      (org-src-font-lock-fontify-block (buffer-substring-no-properties
                      ↪ lang-beg lang-end) (1+ pt) (1- (point)))
                      (font-lock-append-text-property (1+ pt) (1- (point)) 'face
                      ↪ 'org-block)))
                  (font-lock-append-text-property (1- (point)) (point) 'face '(org-block
                  ↪ shadow))
                  (setq pt (point)))))
    (setf pt (point))))

```

```

    (when (and org-prettify-inline-results (re-search-forward "\\= {{{results("
    ↪ limit t))
      (font-lock-append-text-property pt (1+ pt) 'face 'org-block)
      (goto-char pt)))
    (when org-prettify-inline-results
      (goto-char initial-point)
      (org-fontify-inline-src-results limit))))

(defun org-fontify-inline-src-results (limit)
  (while (re-search-forward "{{{results(\\(.+?\\))}}}" limit t)
    (remove-list-of-text-properties (match-beginning 0) (point)
      '(composition
        prettify-symbols-start
        prettify-symbols-end))
    (font-lock-append-text-property (match-beginning 0) (match-end 0) 'face
    ↪ 'org-block)
    (let ((start (match-beginning 0)) (end (match-beginning 1)))
      (with-silent-modifications
        (compose-region start end (if (eq org-prettify-inline-results t) "<" (car
    ↪ org-prettify-inline-results)))
        (add-text-properties start end `(prettify-symbols-start ,start
    ↪ prettify-symbols-end ,end))))
    (let ((start (match-end 1)) (end (point)))
      (with-silent-modifications
        (compose-region start end (if (eq org-prettify-inline-results t) ">" (cdr
    ↪ org-prettify-inline-results)))
        (add-text-properties start end `(prettify-symbols-start ,start
    ↪ prettify-symbols-end ,end))))))

(defun org-fontify-inline-src-blocks-enable ()
  "Add inline src fontification to font-lock in Org.
  Must be run as part of `org-font-lock-set-keywords-hook'."
  (setq org-font-lock-extra-keywords
    (append org-font-lock-extra-keywords '((org-fontify-inline-src-blocks)))))

(add-hook 'org-font-lock-set-keywords-hook #'org-fontify-inline-src-blocks-enable)

```

Doom theme's extra fontification is more problematic than helpful.

```
(setq doom-themes-org-fontify-special-tags nil)
```

## 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.



```
(after! org-superstar
  (setq 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    "☐"
    :checkboxedbox "☑"
    :list_property "☐"
    :em_dash     "—"
    :ellipses    "... "
    :arrow_right "➔"
    :arrow_left  "➡"
    :title       "☐"
    :subtitle    "☐"
    :author      "☐"
    :date        "☐"
    :property    "☐"
    :options     "☐"
    :startup     "☐"
    :macro       "☐"
    :html_head   "☐"
    :html        "☐"
    :latex_class "☐"
    :latex_header "☐"
    :beamer_header "☐"
    :latex       "☐"
    :attr_latex  "☐"
    :attr_html   "☐"
    :attr_org    "☐"
    :begin_quote "☐"
    :end_quote   "☐"
    :caption     "☐"
    :header      "☐"
    :results     "☐"))
```

```

:begin_export  "¿"
:end_export    "¿"
:properties    "¿"
:end           "¿"
:priority_a    ,(proptize "¿" 'face 'all-the-icons-red)
:priority_b    ,(proptize "¿" 'face 'all-the-icons-orange)
:priority_c    ,(proptize "¿" 'face 'all-the-icons-yellow)
:priority_d    ,(proptize "¿" 'face 'all-the-icons-green)
:priority_e    ,(proptize "¿" 'face 'all-the-icons-blue))
(set-ligatures! 'org-mode
:merge t
:checkbox       "[ ]"
:pending      "[-]"
:checkedbox   "[X]"
:list_property ":@"
:em_dash      "---"
:ellipsis     "... "
:arrow_right  "->"
:arrow_left   "<-"
:title        "#+title:"
:subtitle     "#+subtitle:"
:author       "#+author:"
:date         "#+date:"
:property     "#+property:"
:options      "#+options:"
:startup      "#+startup:"
:macro        "#+macro:"
:html_head    "#+html_head:"
:html         "#+html:"
:latex_class  "#+latex_class:"
:latex_header "#+latex_header:"
:beamer_header "#+beamer_header:"
:latex        "#+latex:"
:attr_latex   "#+attr_latex:"
:attr_html    "#+attr_html:"
:attr_org     "#+attr_org:"
: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 "¿")

```

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")
```

```

;; (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))

```

## LaTeX Fragments

**Prettier highlighting** First off, we want those fragments to look good.

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

However, by using native highlighting the `org-block` face is added, and that doesn't look too great — particularly when the fragments are previewed.

Ideally `org-src-font-lock-fontify-block` wouldn't add the `org-block` face, but we can avoid advising that entire function by just adding another face with `:inherit default` which will override the background colour.

Inspecting `org-do-latex-and-related` shows that `"latex"` is the language argument passed, and so we can override the background as discussed above.

```
(require 'org-src)
(add-to-list 'org-src-block-faces '("latex" (:inherit default :extend t)))
```

**More eager rendering** What's better than syntax-highlighted  $\text{\LaTeX}$  is *rendered*  $\text{\LaTeX}$  though, and we can have this be performed automatically with `org-fragtog`.

```
(package! org-fragtog :pin "479e0a1c3610dfe918d89a5f5a92c8aec37f131d")
```

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

**Prettier rendering** It's nice to customise the look of  $\text{\LaTeX}$  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. I'd also like to use some of the functionality from `bmc-maths`, so we'll load that too.

```
(setq org-format-latex-header "\\documentclass{article}
  \\usepackage[usenames]{xcolor}
  \\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}
  ")
```

Since we can, instead of making the background colour match the default face, let's make it transparent.

```
(setq org-format-latex-options
  (plist-put org-format-latex-options :background "Transparent"))
```

**Rendering speed tests** 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: dvi png and dvi svgm. 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  $\text{\LaTeX}$  fragments in-Emacs, but dvisvgm for Section 5.3.6.

### ❖ Warning

Unfortunately, it seems that SVG sizing is annoying ATM, so let's actually not do this right now.

### 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)))
    (cl-incf counter)
    (cons begin counter))
  ((string-match "\\\begin{align}" env)
    (prog2
      (cl-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
        (cl-incf counter (count-matches "\\$"))
        ;; unless there are nonumbers.
        (goto-char (point-min))
        (cl-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
      # border styles
      set tics out nomirror
      set border 3
      # 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))

```

### 5.3.5 Exporting

#### General settings

By default Org only exports the first three levels of headings as ... headings. This is rather unfortunate as my documents frequently stray far beyond three levels of depth. The two main formats I care about exporting to are  $\text{\LaTeX}$  and  $\text{HTML}$ . When using an article class,  $\text{\LaTeX}$  headlines go from  $\text{\section}$ ,  $\text{\subsection}$ ,  $\text{\subsubsection}$ , and  $\text{\paragraph}$  to  $\text{\subgraph}$  — *five* levels.  $\text{HTML5}$  has six levels of headings ( $\text{<h1>}$  to  $\text{<h6>}$ ), but first level Org headings get exported as  $\text{<h2>}$  elements — leaving *five* usable levels.

As such, it would seem to make sense to recognise the first *five* levels of Org headings when exporting.

```
(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))
```

Since I (roughly) track Org HEAD, it makes sense to include the git version in the creator string.

```
(setq org-export-creator-string
      (format "Emacs %s (Org mode %s %s)" emacs-version (org-release)
              (org-git-version)))
```

## Acronym formatting

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).

This has to be implemented on a per-format basis, currently HTML and LaTeX exports are supported.

```
(defun org-export-filter-text-acronym (text backend _info)
  "Wrap suspected acronyms in acronyms-specific formatting.
  Treat sequences of 2+ capital letters (optionally succeeded by \"s\") as an
  acronym.
  Ignore if preceeded by \";\" (for manual prevention) or \"\\\" (for LaTeX
  ↪ commands).

  TODO abstract backend implementations."
  (let ((base-backend
        (cond
          ((org-export-derived-backend-p backend 'latex) 'latex)
          ;; Markdown is derived from HTML, but we don't want to format it
          ((org-export-derived-backend-p backend 'md) nil)
          ((org-export-derived-backend-p backend 'html) 'html)))
        (case-fold-search nil))
    (when base-backend
      (replace-regexp-in-string
       "[;\\\\\\]?\\b[A-Z][A-Z]+s?\\(?:[A-Za-z]\\\\\\\\b\\\\\\\\)"
       (lambda (all-caps-str)
         (cond ((equal (aref all-caps-str 0) ?\\) all-caps-str) ; don't
               ↪ format LaTeX commands
```

```

    ((equal (aref all-caps-str 0) ?\;) (substring all-caps-str 1)) ; just
    ↪ remove not-acronym indicator char ";"
    (t (let* ((final-char (if (string-match-p "[^A-Za-z]" (substring
    ↪ all-caps-str -1 (length all-caps-str)))
        (substring all-caps-str -1 (length
        ↪ all-caps-str))
        nil)) ; needed to re-insert the [^A-Za-z] at
        ↪ the end
      (trailing-s (equal (aref all-caps-str (- (length
        ↪ all-caps-str) (if final-char 2 1))) ?s))
      (acr (if final-char
        (substring all-caps-str 0 (if trailing-s -2 -1))
        (substring all-caps-str 0 (+ (if trailing-s -1 (length
        ↪ all-caps-str)))))
      (pcase base-backend
        ('latex (concat "\\acr{" (s-downcase acr) "}" (when trailing-s
        ↪ "\\acrs{")) final-char))
        ('html (concat "<span class='acr'" acr "</span>" (when
        ↪ trailing-s "<small>s</small>" final-char)))))
    text t t)))

(add-to-list 'org-export-filter-plain-text-functions
  #'org-export-filter-text-acronym)

;; We won't use `org-export-filter-headline-functions' because it
;; passes (and formats) the entire section contents. That's no good.

(defun org-html-format-headline-acronymised (todo todo-type priority text tags info)
  "Like `org-html-format-headline-default-function', but with acronym formatting."
  (org-html-format-headline-default-function
    todo todo-type priority (org-export-filter-text-acronym text 'html info) tags
    ↪ info))
(setq org-html-format-headline-function #'org-html-format-headline-acronymised)

(defun org-latex-format-headline-acronymised (todo todo-type priority text tags info)
  "Like `org-latex-format-headline-default-function', but with acronym formatting."
  (org-latex-format-headline-default-function
    todo todo-type priority (org-export-filter-text-acronym text 'latex info) tags
    ↪ info))
(setq org-latex-format-headline-function #'org-latex-format-headline-acronymised)

```

### 5.3.6 HTML Export

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-collapsible'
  - `org-html-block-collapsible'
  - `org-html-table-wrapped'
  - `org-html--format-toc-headline-collapseable'
  - `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 a `(after! ox-html ...)` block.

```
(after! ox-html
  <<ox-html-conf>>
)
```

### 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-export-in-progress))
      (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
        (if html5-fancy
          "<header class=\"page-header\">%s<h1
            ↪ class=\"title\">%s</h1>\n%s</header>"
          "<h1 class=\"title\">%s</h1>\n")
        (if (or (plist-get info :with-date)
              (plist-get info :with-author))
          (concat "<div class=\"page-meta\">"
            (when (plist-get info :with-date)
              (org-export-data (plist-get info :date) info))
            (when (and (plist-get info :with-date) (plist-get info
              ↪ :with-author)) ", ")
            (when (plist-get info :with-author)
              (org-export-data (plist-get info :author) info))
            "</div>\n")
          "")
        (org-export-data title info)
        (if subtitle
          (format
            (if html5-fancy
              "<p class=\"subtitle\" role=\"doc-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.

```

(defvar org-html-meta-tags-opengraph-image
  '(:image "https://tecosaur.com/resources/org/nib.png"
    :type "image/png"
    :width "200"
    :height "200"
    :alt "Green fountain pen nib")
  "Plist of og:image:PROP properties and their value, for use in
↪ `org-html-meta-tags-fancy'." )

(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))))))

(append
 (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))))
 (when org-html-meta-tags-opengraph-image
  (list (list "property" "og:image" (plist-get org-html-meta-tags-opengraph-image
        ↪ :image))
        (list "property" "og:image:type" (plist-get
        ↪ org-html-meta-tags-opengraph-image :type))
        (list "property" "og:image:width" (plist-get
        ↪ org-html-meta-tags-opengraph-image :width))
        (list "property" "og:image:height" (plist-get
        ↪ org-html-meta-tags-opengraph-image :height))
        (list "property" "og:image:alt" (plist-get
        ↪ org-html-meta-tags-opengraph-image :alt))))
 (list
  (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%TZ"
         (or
          (when-let ((date-str (cadar (org-collect-keywords ("DATE")))))
            (unless (string= date-str (format-time-string "%F"))
              (ignore-errors (encode-time (org-parse-time-string date-str))))
          (if buffer-file-name
              (file-attribute-modification-time (file-attributes
        ↪ buffer-file-name))
              (current-time))))))
  (when buffer-file-name
    (list "property" "og:article:modified_time"

```

```

(format-time-string "%FT%T%Z" (file-attribute-modification-time
  ↪ (file-attributes buffer-file-name)))))))))

(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](https://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.min.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)
```

### Collapsible 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-collapsible (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-export-in-progress))
      (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='copyPreToClipboard(this)'>?</button>
  </div>
  %s
</details>"

ref
(if collapsed-p "" " open")
(if name " class='named'" "")
(concat
  (when name (concat "<span class=\"name\">" name "</span>"))
  "<span class=\"lang\">" lang "</span>")
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-collapsible (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-export-in-progress))
      (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='copyPreToClipboard(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-collapsible)
(advice-add 'org-html-fixed-width :around #'org-html-block-collapsible)
(advice-add 'org-html-property-drawer :around #'org-html-block-collapsible)

```

## Include extra font-locking in htmlize

Org uses [htmlize.el](https://www.gnu.org/software/htmlize/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.

```
(autoload #'highlight-numbers--turn-on "highlight-numbers")
(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 add 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-export-in-progress))
      (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-collapsible (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 collapsable tree."
  :around #'org-html--format-toc-headline
  (if (or (not org-fancy-html-export-mode) (bound-and-true-p
    ↪ org-msg-export-in-progress))
      (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 *ifno children...* logic in `org-html--format-toc-headline-collapsible`. 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-export-in-progress))
      (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

### Extra special strings

The `org-html-special-string-regexps` variable defines substitutions for:

- `\-`, a shy hyphen
- `---`, an em dash
- `--`, an en dash
- `...`, (horizontal) ellipses

However I think it would be nice if there was also a substitution for left/right arrows (`->` and `<-`). This is a `defconst`, but as you may tell from the amount of advice in this config, I'm not above

messing with things I'm not 'supposed' to.

The only minor complication is that `<` and `>` are converted to `&lt;` and `&gt;` before this stage of output processing.

```
(pushnew! org-html-special-string-regexps
  '("&gt;" . "&#8594;")
  '("&lt;-" . "&#8592;"))
```

## 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)
             (not (org-export-derived-backend-p backend 're-reveal))
             org-fancy-html-export-mode)
    (unless (bound-and-true-p org-msg-export-in-progress)
      (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)
```

## Link previews

Sometimes it's nice to make a link particularly prominent, an embed/preview like Twitter does would be nice I think.

We can do this without too much trouble by adding a new link type ever so slightly different from `https` — `Https`.

```
(org-link-set-parameters "Https"
  :follow (lambda (url arg) (browse-url (concat "https:" url)
    ↪ arg))
  :export #'org-url-fancy-export)
```

Then, if we can fetch a plist of the form `(:title "...":description "...":image "...")` for such links via a function `org-url-unfurl-metadata`, we can make a fancy export.

```
(defun org-url-fancy-export (url _desc backend)
  (let ((metadata (org-url-unfurl-metadata (concat "https:" url))))
    (cond
      ((org-export-derived-backend-p backend 'html)
       (concat
        "<div class=\"link-preview\">"
        (format "<a href=\"%s\">" (concat "https:" url))
        (when (plist-get metadata :image)
          (format "<img src=\"%s\"/>" (plist-get metadata :image)))
        "<small>"
        (replace-regexp-in-string "//\\(?:www\\.\\|)?\\([\\^/]+\\)/?.*" "\\1" url)
        "</small><p>"
        (when (plist-get metadata :title)
          (concat "<b>" (org-html-encode-plain-text (plist-get metadata :title))
                  ↪ "</b><br>"))
        (when (plist-get metadata :description)
          (org-html-encode-plain-text (plist-get metadata :description)))
        "</p></a></div>"))
      (t url))))
```

Now we just need to actually implement that metadata extraction function.

```
(setq org-url-unfurl-metadata--cache nil)
(defun org-url-unfurl-metadata (url)
  (cdr (or (assoc url org-url-unfurl-metadata--cache)
    (car (push
      (cons
        url
        (let* ((head-data
          (-filter #'listp
            (cdaddr
              (with-current-buffer (progn (message "Fetching
                ↪ metadata from %s" url)
                ↪ (url-retrieve-synchronously
                ↪ url t t 5))
              (goto-char (point-min))
              (delete-region (point-min) (- (search-forward
                ↪ "<head") 6))
              (delete-region (search-forward "</head>")
                ↪ (point-max))
              (goto-char (point-min))
              (while (re-search-forward
                ↪ "<script[^\u2800]+?</script>" nil t)
                (replace-match ""))
              (goto-char (point-min))
              (while (re-search-forward
                ↪ "<style[^\u2800]+?</style>" nil t)
```

```

        (replace-match ""))
      (libxml-parse-html-region (point-min)
        ↪ (point-max))))))
    (meta (delq nil
      (mapcar
        (lambda (tag)
          (when (eq 'meta (car tag))
            (cons (or (cdr (assoc 'name (cadr tag)))
              (cdr (assoc 'property (cadr
                ↪ tag))))
              (cdr (assoc 'content (cadr tag))))))
          head-data))))
    (let ((title (or (cdr (assoc "og:title" meta))
      (cdr (assoc "twitter:title" meta))
      (nth 2 (assq 'title head-data))))
      (description (or (cdr (assoc "og:description" meta))
        (cdr (assoc "twitter:description" meta))
        (cdr (assoc "description" meta))))
      (image (or (cdr (assoc "og:image" meta))
        (cdr (assoc "twitter:image" meta)))))
      (when image
        (setq image (replace-regexp-in-string
          "^/" (concat "https://" (replace-regexp-in-string
            ↪ "//\\([^\s]+\\)/?.*" "\\1" url) "/"
          (replace-regexp-in-string
            "^/" "https://"
            image))))
        (list :title title :description description :image image)))
    org-url-unfurl-metadata--cache))))

```

## LaTeX Rendering

**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)
```

**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 by 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>")
```

### 5.3.7 $\text{\LaTeX}$ Export

#### Compiling

By default Org uses `pdflatex` + `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 -f -pdf -%latex -shell-escape
↪ -interaction=nonstopmode -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.

## Nicer checkboxes

We'll assume that thanks to the clever preamble the various custom `\checkbox . . .` commands below are defined.

```
(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" "\\\\checkboxUnchecked")
            ("boxminus" "\\\\checkboxTransitive")
            ("boxtimes" "\\\\checkboxChecked")
            (_ (substring fullmatch 9 -3))) "]" )
        text)))

(add-to-list 'org-export-filter-item-functions
  '+org-export-latex-fancy-item-checkboxes)
```

## Class templates

I really like the KOMA bundle. It provides a set of mechanisms to tweak document styling which is both easy to use, and quite comprehensive. For example, I rather like section numbers in the margin, which can be accomplished with

```

\renewcommand\sectionformat{\llap{\thesecion\autodot\enskip}}
\renewcommand\subsectionformat{\llap{\thesubsection\autodot\enskip}}
\renewcommand\subsubsectionformat{\llap{\thesubsubsection\autodot\enskip}}

```

It can also be nice to have big \chapters.

```
\RedeclareSectionCommand[afterindent=false, beforeskip=0pt, afterskip=0pt,  
  ↪ innerskip=0pt]{chapter}  
\setkomafont{chapter}{\normalfont\Huge}
```

```

\\renewcommand*{\\chapterheadstartvskip}{\\vspace*{0\\baselineskip}}
\\renewcommand*{\\chapterheadendvskip}{\\vspace*{0\\baselineskip}}
\\renewcommand*{\\chapterformat}{%
  \\fontsize{60}{30}\\selectfont\\rlap{\\hspace{6pt}\\thechapter}}
\\renewcommand*{\\chapterlinesformat[3]{%
  \\parbox[b]{\\dimexpr\\textwidth-0.5em\\relax}{%
    \\raggedleft{\\large\\scshape\\bfseries\\chapapp}\\vspace{-0.5ex}\\par\\Huge#3}}%
    \\hfill\\makebox[0pt][l]{#2}}

```

Now let's just sprinkle some KOMA all over the Org  $\text{\LaTeX}$  classes.

```

(after! ox-latex
  (let* ((article-sections '("\\section{%s}" . "\\section*{%s}")
        ("\\subsection{%s}" . "\\subsection*{%s}")
        ("\\subsubsection{%s}" . "\\subsubsection*{%s}")
        ("\\paragraph{%s}" . "\\paragraph*{%s}")
        ("\\subparagraph{%s}" . "\\subparagraph*{%s}"))
        (book-sections (append '("\\chapter{%s}" . "\\chapter*{%s}")
                                article-sections))
        (hanging-secnum-preamble "
<<latex-hanging-secnum>>
")
        (big-chap-preamble "
<<latex-big-chapter>>
"))
    (setcdr (assoc "article" org-latex-classes)
      `(,(concat "\\documentclass{scrartcl}" hanging-secnum-preamble)
        ,@article-sections))
    (add-to-list 'org-latex-classes
      `("report" ,(concat "\\documentclass{scrartcl}"
        ↪ hanging-secnum-preamble)
        ,@article-sections))
    (add-to-list 'org-latex-classes
      `("book" ,(concat "\\documentclass[twoside=false]{scrbook}"
        ↪ big-chap-preamble hanging-secnum-preamble)
        ,@book-sections))
    (add-to-list 'org-latex-classes
      `("blank" "[NO-DEFAULT-PACKAGES]\\n[NO-PACKAGES]\\n[EXTRA]"
        ,@article-sections))
    (add-to-list 'org-latex-classes
      `("bmc-article"
        ↪ "\\documentclass[article,code,maths]{bmc}\\n[NO-DEFAULT-PACKAGES]\\n[NO-PACKAGES]\\n[EXTRA]"
        ,@article-sections))
    (add-to-list 'org-latex-classes
      `("bmc"
        ↪ "\\documentclass[code,maths]{bmc}\\n[NO-DEFAULT-PACKAGES]\\n[NO-PACKAGES]\\n[EXTRA]"
        ,@book-sections))))

```

```
(setq org-latex-tables-booktabs t
      org-latex-hyperref-template "
<<latex-fancy-hyperref>>
"
      org-latex-reference-command "\\cref{%s}")
```

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}
```

### A cleverer preamble

**Use case** We often want particular snippets of  $\text{\LaTeX}$  in our documents preambles. It's a pain to have to work out / remember them every time. For example, I almost always want to include the snippet below.

```
\\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 provide some granularity by splitting up the features we want, and then take the experience to a whole new level by implementing a system to automatically detect which features are desired and generating a preamble that provides these features.

**Conditional Content** Let's consider content we want in particular situations.



Captions could do with a bit of tweaking such that

- You can easily have multiple captions
- Links to figures take you to the *top* of the figure (not the bottom)
- Caption labels could do with being emphasised slightly more
- Multiline captions should run ragged-right, but only when then span more than one line

```
\usepackage{subcaption}
\usepackage[hypcap=true]{caption}
\setkomafont{caption}{\sffamily\small}
\setkomafont{captionlabel}{\upshape\bfseries}
\captionsetup{justification=raggedright,singlelinecheck=true}
\usepackage{capt-of} % required by Org
```

The default checkboxes look rather ugly, so let's provide some prettier alternatives.

```
\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$}
```

It's nice to have “message blocks”, things like info/warning/error/success. A  $\text{\LaTeX}$  macro should make them trivial to create.

```
% args = #1 Name, #2 Colour, #3 Ding, #4 Label
\newcommand{\defsimplebox}[4]{%
  \definecolor{#1}{HTML}{#2}
  \newenvironment{#1}[1]{}
  {%
    \par\vspace{-0.7\baselineskip}%
    \textcolor{#1}{#3}
    ↪ \textcolor{#1}{\textbf{\def\temp{##1}\ifx\temp\empty#4\else#1\fi}}%
    \vspace{-0.8\baselineskip}
    \begin{addmargin}[1em]{1em}
  }{%
    \end{addmargin}
    \vspace{-0.5\baselineskip}
  }%
}
```

Lastly, we will pass this content into some global variables we for ease of access.

```
(defvar org-latex-embed-files-preamble "
<<org-latex-embed-files-preamble>>
"
"Preamble that embeds files within the pdf.")
```

```
(defvar org-latex-caption-preamble "
<<org-latex-caption-preamble>>
"
  "Preamble that improves captions.")

(defvar org-latex-checkbox-preamble "
<<org-latex-checkbox-preamble>>
"
  "Preamble that improves checkboxes.")

(defvar org-latex-box-preamble "
<<org-latex-box-preamble>>
"
  "Preamble that provides a macro for custom boxes.")
```

In the “universal preamble”, we already embed the source .org file, but it would be nice to embed all the tangled files. This is fairly easy to accomplish by mapping each tangled file to a form which embeds the file if it exists. Considering we’re going this far, why not add a dedicated `#+embed` keyword, so we can embed whatever we want.

```
(defun org-latex-embed-extra-files ()
  "Return a string that uses embedfile to embed all tangled files."
  (mapconcat
    (lambda (file-desc)
      (format "\\IfFileExists{%1$s}{\\embedfile[desc=%2$s]{%1$s}}{ }"
        (thread-last (car file-desc)
          (replace-regexp-in-string "\\\\" "\\\\")
          (replace-regexp-in-string "~" "\\string~"))
        (cdr file-desc)))
    (append
      (mapcar (lambda (f-block)
        (let ((file-lang (cons (or (car f-block) (caddr (cadr f-block))) (caadr
          ↪ f-block))))
          (cons (car file-lang) (format "Tangled %s file" (cdr file-lang)))))
        (org-babel-tangle-collect-blocks)) ; all files being tangled to
      (let (extra-files)
        (save-excursion
          (goto-char (point-min))
          (while (re-search-forward "^[ \t]*#+embed:" nil t)
            (let* ((file-desc (split-string (org-element-property :value
              ↪ (org-element-at-point)) " :desc\\(?:ription\\)? "))
              (push (cons (car file-desc) (or (cdr file-desc) "Extra file"))
                ↪ extra-files))))
            (nreverse extra-files)))
        "\\n"))
```

Now all tangled files will be embedded, and we can embed arbitrary files like so:

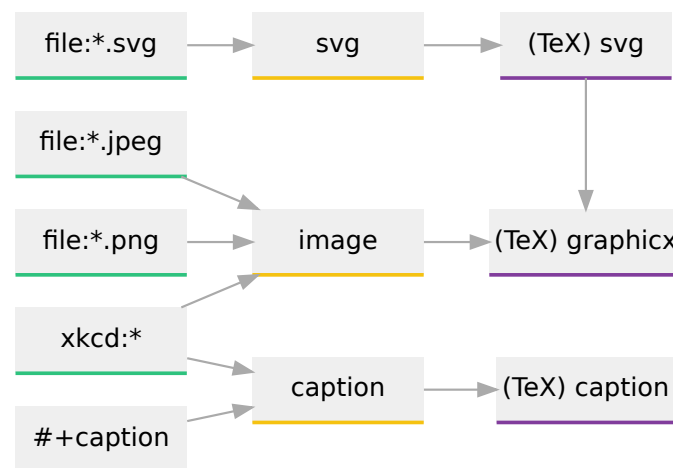
```
#+embed: some-file :description flavour text about the file
```

This currently won't complete or anything like that, as we haven't told Org that it's a keyword yet. It's also  $\text{\LaTeX}$ -specific, so maybe it should be changed to `#+latex_embed` or something like that.

**Content-feature-preamble association** Initially this idea was implemented with an alist that associated a construct that would search the current Org file for an indication that some feature was needed, with a  $\text{\LaTeX}$  snippet to be inserted in the preamble which would provide that feature. This is all well and good when there is a bijection between detected features and the  $\text{\LaTeX}$  code needed to support those features, but in many cases this relation is not injective.

To better model the reality of the situation, I add an extra layer to this process where each detected feature gives a list of required “feature flags”. Simply by merging the lists of feature flags we no longer have to require injectivity to avoid  $\text{\LaTeX}$  duplication. Then the extra layer forms a bijection between these feature flags and a specification which can be used to implement the feature.

This model also provides a number of nice secondary benefits, such as a simple implementation of feature dependency.



**Figure 5.1:** Association between Org features, feature flags, and  $\text{\LaTeX}$  snippets required.

First we will implement the feature detection component of this model. I'd like this to be able to use as much state information as possible, so the feature tests should be very versatile.

```
(defvar org-latex-embed-files t
  "Embed the source .org, .tex, and any tangled files.")
(defvar org-latex-use-microtype t
  "Use the microtype package.")
(defvar org-latex-italic-quotes t
```

[illegible]

Then we provide a way to generate the preamble that provides those features. In addition to the

features named in `org-latex-conditional-features` we'll also create *meta-features*, which can be required by other features (with `:requires`), or be active by default (`:eager t`). For further control I some features may only be used when certain other features are active (with `:when`), and masked by other features (with `:prevents`). I will use the convention of starting meta-features with `.`, and `:eager` features with `!` to make their nature more readily apparent.

Another consideration in  $\text{\LaTeX}$  is load order, which matters in some cases. Beyond that, it's nice to have some sort of sensible ordering. For this I'll introduce an `:order` keyword. Using this I'll arrange snippets as follows.

- -2 Embed files setup
- -1 Extra file embedding
- 0 Typography
  - 0 Fonts themselves
  - 0.1 Typographic tweaks (microtype)
  - 0.2 Maths setup
  - 0.3 Maths font
  - 0.4 Extra text shaping (`\acr`)
  - 0.5–0.9 Miscellaneous text modifications, trying to put shorter snippets first
- 1 (*default*)
- 2 Tables and figures
- 3 Miscellaneous short content
- 4 Fancy boxes

```
(defvar org-latex-feature-implementations
  '( (image      :snippet "\\usepackage{graphicx}" :order 2)
    (svg         :snippet "\\usepackage[inkscapelatex=false]{svg}" :order 2)
    (maths       :snippet "\\usepackage[nofont]{bmc-maths}" :order 0.2)
    (table       :snippet "\\usepackage{longtable}\n\\usepackage{booktabs}" :order
      ↪ 2)
    (cleveref     :snippet "\\usepackage[capitalize]{cleveref}" :order 1) ; after
    ↪ bmc-maths
    (underline    :snippet "\\usepackage[normalem]{ulem}" :order 0.5)
    (float-wrap   :snippet "\\usepackage{wrapfig}" :order 2)
    (rotate       :snippet "\\usepackage{rotating}" :order 2)
    (caption      :snippet org-latex-caption-preamble :order 2.1)
    (microtype    :snippet
      ↪ "\\usepackage[activate={true,nocompatibility},final,tracking=true,kerning=true,spacing=true,factor
      ↪ :order 0.1)
    (embed-files  :snippet org-latex-embed-files-preamble :order -2)
```

```

(embed-tangled :requires embed-files :snippet (concat
  ↪ (org-latex-embed-extra-files) "\n") :order -1)
(acronym       :snippet "\newcommand{\acr}[1]{\protect\textls*[10]{\scshape
  ↪ #1}}\n\newcommand{\acrs}{\protect\scalebox{.91}[.84]{\hspace{0.15ex}s}}"
  ↪ :order 0.4)
(italic-quotes :snippet
  ↪ "\renewcommand{\quote}{\list{}{\rightmargin\leftmargin}\item\relax\em}\n"
  ↪ :order 0.5)
(par-sep       :snippet
  ↪ "\setlength{\parskip}{\baselineskip}\n\setlength{\parindent}{0pt}\n"
  ↪ :order 0.5)
(.pifont       :snippet "\usepackage{pifont}")
(checkbox      :requires .pifont :order 3
  :snippet (concat (unless (memq 'maths features)
    "\usepackage{amssymb} % provides \square")
    org-latex-checkbox-preamble))
(.fancy-box    :requires .pifont :snippet org-latex-box-preamble :order 3.9)
(box-warning   :requires .fancy-box :snippet
  ↪ "\defsimplebox{warning}{e66100}{\ding{68}}{Warning}" :order 4)
(box-info      :requires .fancy-box :snippet
  ↪ "\defsimplebox{info}{3584e4}{\ding{68}}{Information}" :order 4)
(box-notes     :requires .fancy-box :snippet
  ↪ "\defsimplebox{notes}{26a269}{\ding{43}}{Notes}" :order 4)
(box-success   :requires .fancy-box :snippet
  ↪ "\defsimplebox{success}{26a269}{\ding{68}}{\vspace{-\baselineskip}}"
  ↪ :order 4)
(box-error     :requires .fancy-box :snippet
  ↪ "\defsimplebox{error}{c01c28}{\ding{68}}{Important}" :order 4))
"LaTeX features and details required to implement them.
  List where the car is the feature symbol, and the rest forms a plist with the
  following keys:
  - :snippet, which may be either
    - a string which should be included in the preamble
    - a symbol, the value of which is included in the preamble
    - a function, which is evaluated with the list of feature flags as its
      single argument. The result of which is included in the preamble
    - a list, which is passed to `eval', with a list of feature flags available
      as "features"
  - :requires, a feature or list of features that must be available
  - :when, a feature or list of features that when all available should cause this
    to be automatically enabled.
  - :prevents, a feature or list of features that should be masked
  - :order, for when ordering is important. Lower values appear first.
    The default is 0.
  Features that start with ! will be eagerly loaded, i.e. without being
  ↪ detected.")

```

**Feature determination** Now that we have `org-latex-conditional-features` defined, we need to use it to extract a list of features found in an Org buffer.

```
(defun org-latex-detect-features (&optional buffer info)
  "List features from `org-latex-conditional-features' detected in BUFFER."
  (let ((case-fold-search nil))
    (with-current-buffer (or buffer (current-buffer))
      (delete-dups
       (mapcan (lambda (construct-feature)
                 (when (let ((out (pcase (car construct-feature)
                                         ((pred stringp) (car construct-feature))
                                         ((pred functionp) (funcall (car construct-feature)
                                                                    ↪ info))
                                         ((pred listp) (eval (car construct-feature)))
                                         ((pred symbolp) (symbol-value (car
                                                                    ↪ construct-feature))))
                               (_ (user-error "org-latex-conditional-features key
                                                                    ↪ %s unable to be used" (car
                                                                    ↪ construct-feature))))))
               (if (stringp out)
                   (save-excursion
                     (goto-char (point-min))
                     (re-search-forward out nil t))
                   out))
               (if (listp (cdr construct-feature)) (cdr construct-feature) (list
                                                                    ↪ (cdr construct-feature))))))
              org-latex-conditional-features))))))
```

**Preamble generation** Once a list of required features has been determined, we want to use `org-latex-feature-implementations` to generate the  $\text{\LaTeX}$  which should be inserted into the preamble to provide those features.

First we want to process our fancy keywords in `org-latex-feature-implementations` to produce an *expanded* list of features. We'll do that by performing the following steps.

- The dependencies for each listed feature are added to feature list (`:requires`).
- The `:when` conditions of each feature, and available features with `:eager` `t`, are evaluated, and added/removed accordingly
- Any features present in a `:prevents` value are removed
- The feature list is scrubbed of duplicates
- The feature list is sorted by `:order` (ascending)

```
(defun org-latex-expand-features (features)
```

```

"For each feature in FEATURES process :requires, :when, and :prevents keywords and
↪ sort according to :order."
(dolist (feature features)
  (unless (assoc feature org-latex-feature-implementations)
    (error "Feature %s not provided in org-latex-feature-implementations" feature)))
(setq current features)
(while current
  (when-let ((requirements (plist-get (cdr (assoc (car current)
    ↪ org-latex-feature-implementations)) :requires)))
    (setcdr current (if (listp requirements)
      (append requirements (cdr current))
      (cons requirements (cdr current)))))
    (setq current (cdr current)))
(dolist (potential-feature
  (append features (delq nil (mapcar (lambda (feat)
    (when (plist-get (cdr feat) :eager)
      (car feat)))
    org-latex-feature-implementations))))
  (when-let ((prerequisites (plist-get (cdr (assoc potential-feature
    ↪ org-latex-feature-implementations)) :when)))
    (setf features (if (if (listp prerequisites)
      (cl-every (lambda (preq) (memq preq features))
        ↪ prerequisites)
      (memq prerequisites features))
      (append (list potential-feature) features)
      (delq potential-feature features)))))
(dolist (feature features)
  (when-let ((prevents (plist-get (cdr (assoc feature
    ↪ org-latex-feature-implementations)) :prevents)))
    (setf features (cl-set-difference features (if (listp prevents) prevents (list
      ↪ prevents)))))
(sort (delete-dups features)
  (lambda (feat1 feat2)
    (if (< (or (plist-get (cdr (assoc feat1 org-latex-feature-implementations))
      ↪ :order) 1)
      (or (plist-get (cdr (assoc feat2 org-latex-feature-implementations))
        ↪ :order) 1)
      t nil))))

```

Now that we have a nice list of the final features to use, we can just pull out their snippets and join the result together.

```

(defun org-latex-generate-features-preamble (features)
  "Generate the LaTeX preamble content required to provide FEATURES.
This is done according to `org-latex-feature-implementations'"
  (let ((expanded-features (org-latex-expand-features features)))
    (concat
      (format "\\n%% features: %s\\n" expanded-features)

```



```
(mapconcat (lambda (feature)
  (when-let ((snippet (plist-get (cdr (assoc feature
    ↪ org-latex-feature-implementations)) :snippet)))
    (concat
      (pcase snippet
        ((pred stringp) snippet)
        ((pred functionp) (funcall snippet features))
        ((pred listp) (eval `(let ((features ',features))
          ↪ (,@snippet))))
        ((pred symbolp) (symbol-value snippet))
        (_ (user-error "org-latex-feature-implementations :snippet
          ↪ value %s unable to be used" snippet)))
      "\n")))
  expanded-features
  "")
"% end features\n"))
```

Then Org needs to be advised to actually use this generated preamble content.

```
(defvar info--tmp nil)

(defadvice! org-latex-save-info (info &optional t_ s_)
  :before #'org-latex-make-preamble
  (setq info--tmp info))

(defadvice! org-splice-latex-header-and-generated-preamble-a (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
        (org-latex-generate-features-preamble (org-latex-detect-features nil
          ↪ info--tmp))
        "\n")))))
```

My use of `info--tmp` is somewhat hacky. When I try to upstream this, this should become much cleaner as I can pass `info` through by directly modifying `org-latex-make-preamble`.

**Reduce default packages** Thanks to our additions, we can remove a few packages from `org-latex-default-packages`

There are also some obsolete entries in the default value, specifically

- `grffile`'s capabilities are built into the current version of `graphicx`
- `textcomp`'s functionality has been included in LaTeX's core for a while now

```
(setq org-latex-default-packages-alist
  '(("AUTO" "inputenc" t ("pdflatex"))
    ("T1" "fontenc" t ("pdflatex"))
    ("" "xcolor" nil) ; Generally useful
    ("" "hyperref" nil)))
```

## Font collections

Using the lovely conditional preamble, I'll define a number of font collections that can be used for  $\text{\LaTeX}$  exports. Who knows, maybe I'll use it with other export formats too at some point.

To start with I'll create a default state variable and register fontset as part of #+options.

```
(defvar org-latex-default-fontset 'alegreya
  "Fontset from `org-latex-fontsets' to use by default.
  As cm (computer modern) is TeX's default, that causes nothing
  to be added to the document.
  If \"nil\" no custom fonts will ever be used.")

(eval '(cl-pushnew '(:latex-font-set nil "fontset" org-latex-default-fontset)
  (org-export-backend-options (org-export-get-backend 'latex))))
```

Then a function is needed to generate a  $\text{\LaTeX}$  snippet which applies the fontset. It would be nice if this could be done for individual styles and use different styles as the main document font. If the individual typefaces for a fontset are defined individually as `:serif`, `:sans`, `:mono`, and `:maths`. I can use those to generate  $\text{\LaTeX}$  for subsets of the full fontset. Then, if I don't let any fontset names have - in them, I can use -sans and -mono as suffixes that specify the document font to use.

```
(defun org-latex-fontset-entry ()
  "Get the fontset spec of the current file.
  Has format \"name\" or \"name-style\" where 'name' is one of
  the cars in `org-latex-fontsets'."
  (let ((fontset-spec
        (symbol-name
         (or (car (delq nil
                        (mapcar
                         (lambda (opt-line)
                           (plist-get (org-export--parse-option-keyword opt-line)
                                       'latex)
                                       :latex-font-set))
                         (cdar (org-collect-keywords '("OPTIONS"))))))
         org-latex-default-fontset))))
    (cons (intern (car (split-string fontset-spec "-")))
          (when (cadr (split-string fontset-spec "-"))
```

```

      (intern (concat ":" (cadr (split-string fontset-spec "-"))))))))

(defun org-latex-fontset (&rest desired-styles)
  "Generate a LaTeX preamble snippet which applies the current fontset for
  ↪ DESIRED-STYLES."
  (let* ((fontset-spec (org-latex-fontset-entry))
        (fontset (alist-get (car fontset-spec) org-latex-fontsets)))
    (if fontset
      (concat
        (mapconcat
          (lambda (style)
            (when (plist-get fontset style)
              (concat (plist-get fontset style) "\n"))))
          desired-styles
          "")
      (when (memq (cdr fontset-spec) desired-styles)
        (pcase (cdr fontset-spec)
          (:serif "\\renewcommand{\\familydefault}{\\rmdefault}\\n")
          (:sans "\\renewcommand{\\familydefault}{\\sfdefault}\\n")
          (:mono "\\renewcommand{\\familydefault}{\\ttdefault}\\n"))))
      (error "Font-set %s is not provided in org-latex-fontsets" (car
        ↪ fontset-spec))))))

```

Now that all the functionality has been implemented, we should hook it into our preamble generation.

```

(add-to-list 'org-latex-conditional-features '(org-latex-default-fontset .
  ↪ custom-font) t)
(add-to-list 'org-latex-feature-implementations '(custom-font :snippet
  ↪ (org-latex-fontset :serif :sans :mono) :order 0) t)
(add-to-list 'org-latex-feature-implementations '(.custom-maths-font :eager t :when
  ↪ (custom-font maths) :snippet (org-latex-fontset :maths) :order 0.3) t)

```

Finally, we just need to add some fonts.

```

(defvar org-latex-fontsets
  '((cm nil) ; computer modern
    (## nil) ; no font set
    (alegreya
      :serif "\\usepackage[osf]{Alegreya}"
      :sans "\\usepackage{AlegreyaSans}"
      :mono "\\usepackage[scale=0.88]{sourcecodepro}"
      :maths "\\usepackage[varbb]{newpxmath}")
    (biolinum
      :serif "\\usepackage[osf]{libertineRoman}"
      :sans "\\usepackage[sfdefault,osf]{biolinum}"
      :mono "\\usepackage[scale=0.88]{sourcecodepro}"
      :maths "\\usepackage[libertine,varvw]{newtxmath}")
    (fira

```

```

:sans "\\usepackage[sfdefault,scale=0.85]{FiraSans}"
:mono "\\usepackage[scale=0.80]{FiraMono}"
:maths "\\usepackage{newtxsf} % change to firamath in future?"
(kp
 :serif "\\usepackage{kpfonts}")
(newpx
 :serif "\\usepackage{newpxtext}"
 :sans "\\usepackage{gillius}"
 :mono "\\usepackage[scale=0.9]{sourcecodepro}"
 :maths "\\usepackage[varbb]{newpxmath}")
(oto
 :serif "\\usepackage[osf]{oto-serif}"
 :sans "\\usepackage[osf]{oto-sans}"
 :mono "\\usepackage[scale=0.96]{oto-mono}"
 :maths "\\usepackage{otomath}")
(plex
 :serif "\\usepackage{plex-serif}"
 :sans "\\usepackage{plex-sans}"
 :mono "\\usepackage[scale=0.95]{plex-mono}"
 :maths "\\usepackage{newtxmath}") ; may be plex-based in future
(source
 :serif "\\usepackage[osf]{sourceserifpro}"
 :sans "\\usepackage[osf]{sourcesanspro}"
 :mono "\\usepackage[scale=0.95]{sourcecodepro}"
 :maths "\\usepackage{newtxmath}") ; may be sourceserifpro-based in future
(times
 :serif "\\usepackage{newtxtext}"
 :maths "\\usepackage{newtxmath}"))

```

*"A list of fontset specifications.  
Each car is the name of the fontset (which cannot include \"-").  
Each cdr is a plist with (optional) keys :serif, :sans, :mono, and :maths.  
A key's value is a LaTeX snippet which loads such a font."*

When we're using *Alegreya* we can apply a lovely little tweak to `tabular` which (locally) changes the figures used to lining fixed-width.

```

(add-to-list 'org-latex-conditional-features '((string= (car
  ↪ (org-latex-fontset-entry)) "alegreya") . alegreya-typeface))
(add-to-list 'org-latex-feature-implementations '(alegreya-typeface) t)
(add-to-list 'org-latex-feature-implementations'(.alegreya-tabular-figures :eager t
  :when (alegreya-typeface table) :order 0.5 :snippet "
  \\makeatletter
  % tabular lining figures in tables
  \\renewcommand{\\tabular}{\\AlegreyaTLF\\let\\@halignto\\@empty\\@tabular}
  \\makeatother\n") t)

```

Due to the *Alegreya*'s metrics, the `\LaTeX` symbol doesn't quite look right. We can correct for this by redefining it with subtly shifted kerning.

```
(add-to-list 'org-latex-conditional-features '("LaTeX" . latex-symbol))
(add-to-list 'org-latex-feature-implementations '(latex-symbol :when alegreya-typeface
:order 0.5 :snippet "
\\makeatletter
% Kerning around the A needs adjusting
\\DeclareRobustCommand{\\LaTeX}{L\\kern-.24em%
  {\\sbox\\z@ T%
    \\vbox to\\ht\\z@{\\hbox{\\check@mathfonts
      \\fontsize\\sf@size\\z@
      \\math@fontsfalse\\selectfont
      A}%
    \\vss}%
  }%
  \\kern-.10em%
  \\TeX}
\\makeatother\\n") t)
```

## Cover page

To make a nice cover page, a simple method that comes to mind is just redefining `\maketitle`. To get precise control over the positioning we'll use the `tikz` package, and then add in the Tikz libraries `calc` and `shapes.geometric` to make some nice decorations for the background.

I'll start off by setting up the required additions to the preamble. This will accomplish the following:

- Load the required packages
- Redefine `\maketitle`
- Draw an Org icon with Tikz to use in the cover page (it's a little easter egg)
- Start a new page after the table of contents by redefining `\tableofcontents`

```
\\usepackage{tikz}
\\usetikzlibrary{shapes.geometric}
\\usetikzlibrary{calc}

\\newsavebox\\orgicon
\\begin{lrbox}{\\orgicon}
  \\begin{tikzpicture}[y=0.80pt, x=0.80pt, inner sep=0pt, outer sep=0pt]
```

```

\\path[fill=black!6] (16.15,24.00) .. controls (15.58,24.00) and (13.99,20.69) ..
→ (12.77,18.06) arc(215.55:180.20:2.19) .. controls (12.33,19.91) and
→ (11.27,19.09) .. (11.43,18.05) .. controls (11.36,18.09) and (10.17,17.83) ..
→ (10.17,17.82) .. controls (9.94,18.75) and (9.37,19.44) .. (9.02,18.39) ..
→ controls (8.32,16.72) and (8.14,15.40) .. (9.13,13.80) .. controls (8.22,9.74)
→ and (2.18,7.75) .. (2.81,4.47) .. controls (2.99,4.47) and (4.45,0.99) ..
→ (9.15,2.41) .. controls (14.71,3.99) and (17.77,0.30) .. (18.13,0.04) ..
→ controls (18.65,-0.49) and (16.78,4.61) .. (12.83,6.90) .. controls
→ (10.49,8.18) and (11.96,10.38) .. (12.12,11.15) .. controls (12.12,11.15) and
→ (14.00,9.84) .. (15.36,11.85) .. controls (16.58,11.53) and (17.40,12.07) ..
→ (18.46,11.69) .. controls (19.10,11.41) and (21.79,11.58) .. (20.79,13.08) ..
→ controls (20.79,13.08) and (21.71,13.90) .. (21.80,13.99) .. controls
→ (21.97,14.75) and (21.59,14.91) .. (21.47,15.12) .. controls (21.44,15.60) and
→ (21.04,15.79) .. (20.55,15.44) .. controls (19.45,15.64) and (18.36,15.55) ..
→ (17.83,15.59) .. controls (16.65,15.76) and (15.67,16.38) .. (15.67,16.38) ..
→ controls (15.40,17.19) and (14.82,17.01) .. (14.09,17.32) .. controls
→ (14.70,18.69) and (14.76,19.32) .. (15.50,21.32) .. controls (15.76,22.37) and
→ (16.54,24.00) .. (16.15,24.00) -- cycle(7.83,16.74) .. controls (6.83,15.71)
→ and (5.72,15.70) .. (4.05,15.42) .. controls (2.75,15.19) and (0.39,12.97) ..
→ (0.02,10.68) .. controls (-0.02,10.07) and (-0.06,8.50) .. (0.45,7.18) ..
→ controls (0.94,6.05) and (1.27,5.45) .. (2.29,4.85) .. controls (1.41,8.02)
→ and (7.59,10.18) .. (8.55,13.80) -- (8.55,13.80) .. controls (7.73,15.00) and
→ (7.80,15.64) .. (7.83,16.74) -- cycle;

\\end{tikzpicture}
\\end{lrbox}

\\makeatletter
\\g@addto@macro\\tableofcontents{\\clearpage}
\\renewcommand\\maketitle{
  \\thispagestyle{empty}
  \\hyphenpenalty=10000 % hyphens look bad in titles
  \\renewcommand{\\baselinestretch}{1.1}
  \\let\\oldtoday\\today
  \\renewcommand{\\today}{\\LARGE\\number\\year\\LARGE%
    \\ifcase \\month \\or Jan\\or Feb\\or Mar\\or Apr\\or May \\or Jun\\or Jul\\or
    Aug\\or Sep\\or Oct\\or Nov\\or Dec\\fi
    ~\\number\\day}
  \\begin{tikzpicture}[remember picture,overlay]
    %% Background Polygons %%
    \\foreach \\i in {2.5,...,22} % bottom left
    {\\node[rounded corners,black!3.5,draw,regular polygon,regular polygon sides=6,
    → minimum size=\\i cm,ultra thick] at ($(current page.west)+(2.5,-4.2)$) {};}
    \\foreach \\i in {0.5,...,22} % top left
    {\\node[rounded corners,black!5,draw,regular polygon,regular polygon sides=6,
    → minimum size=\\i cm,ultra thick] at ($(current page.north west)+(2.5,2)$) {}
    →;}
    \\node[rounded corners,fill=black!4,regular polygon,regular polygon sides=6,
    → minimum size=5.5 cm,ultra thick] at ($(current page.north west)+(2.5,2)$) {};
  \\end{tikzpicture}
}

```

```

\\foreach \\i in {0.5,...,24} % top right
{\\node[rounded corners,black!2,draw,regular polygon,regular polygon sides=6,
  ↪ minimum size=\\i cm,ultra thick] at ($(current page.north east)+(0,-8.5)$) {}
  ↪ };
\\node[fill=black!3,rounded corners,regular polygon,regular polygon sides=6,
  ↪ minimum size=2.5 cm,ultra thick] at ($(current page.north east)+(0,-8.5)$) {};
\\foreach \\i in {21,...,3} % bottom right
{\\node[black!3,rounded corners,draw,regular polygon,regular polygon sides=6,
  ↪ minimum size=\\i cm,ultra thick] at ($(current page.south east)+(-1.5,0.75)$)
  ↪ {} };
\\node[fill=black!3,rounded corners,regular polygon,regular polygon sides=6,
  ↪ minimum size=2 cm,ultra thick] at ($(current page.south east)+(-1.5,0.75)$)
  ↪ {};
\\node[align=center, scale=1.4] at ($(current page.south east)+(-1.5,0.75)$)
  ↪ {\\usebox\\orgicon};
%% Text %%
\\node[left, align=right, black, text width=0.8\\paperwidth, minimum height=3cm,
  ↪ rounded corners,font=\\Huge\\bfseries] at ($(current page.north
  ↪ east)+(-2,-8.5)$)
{\\@title};
\\node[left, align=right, black, text width=0.8\\paperwidth, minimum height=2cm,
  ↪ rounded corners, font=\\Large] at ($(current page.north east)+(-2,-11.8)$)
{\\scshape \\@author};
\\renewcommand{\\baselinestretch}{0.75}
\\node[align=center,rounded corners,fill=black!3,text=black,regular
  ↪ polygon,regular polygon sides=6, minimum size=2.5 cm,inner sep=0,
  ↪ font=\\Large\\bfseries ] at ($(current page.west)+(2.5,-4.2)$)
{\\@date};
\\end{tikzpicture}
\\let\\today\\oldtoday
\\clearpage}
\\makeatother

```

Now we've got a nice cover page to work with, we just need to use it every now and then. Adding this to `#+options` feels most appropriate. Let's have the `coverpage` option accept `auto` as a value and then decide whether or not a cover page should be used based on the word count — I'll have this be the global default. Then we just want to insert a `LATEX` snippet tweak the subtitle format to use the cover page.

```

(defvar org-latex-cover-page 'auto
  "When t, use a cover page by default.
  When auto, use a cover page when the document's wordcount exceeds
  `org-latex-cover-page-wordcount-threshold'.
  Set with #+option: coverpage:{yes,auto,no} in org buffers.")
(defvar org-latex-cover-page-wordcount-threshold 5000
  "Document word count at which a cover page will be used automatically.
  This condition is applied when cover page option is set to auto.")

```

```

(defvar org-latex-subtitle-coverpage-format
  ↪ "\\bigskip\\n\\LARGE\\mdseries\\itshape\\color{black!80} %s\\par"
  "Variant of `org-latex-subtitle-format' to use with the cover page.")
(defvar org-latex-cover-page-maketitle "
<<latex-cover-page>>
"
  "LaTeX snippet for the preamble that sets \\maketitle to produce a cover page.")

(eval '(cl-pushnew '(:latex-cover-page nil "coverpage" org-latex-cover-page)
  (org-export-backend-options (org-export-get-backend 'latex))))

(defun org-latex-cover-page-p ()
  "Whether a cover page should be used when exporting this Org file."
  (pcase (or (car
    (delq nil
      (mapcar
        (lambda (opt-line)
          (plist-get (org-export--parse-option-keyword opt-line 'latex)
            ↪ :latex-cover-page))
        (cdar (org-collect-keywords '("OPTIONS")))))
      org-latex-cover-page)
    ((or 't 'yes) t)
    ('auto (when (> (count-words (point-min) (point-max))
      ↪ org-latex-cover-page-wordcount-threshold) t))
    (_ nil)))

(defadvice! org-latex-set-coverpage-subtitle-format-a (contents info)
  "Set the subtitle format when a cover page is being used."
  :before #'org-latex-template
  (when (org-latex-cover-page-p)
    (setf info (plist-put info :latex-subtitle-format
      ↪ org-latex-subtitle-coverpage-format))))

(add-to-list 'org-latex-feature-implementations '(cover-page :snippet
  ↪ org-latex-cover-page-maketitle :order 9) t)
(add-to-list 'org-latex-conditional-features '((org-latex-cover-page-p) . cover-page)
  ↪ t)

```

## Condensed lists

$\text{\LaTeX}$  is generally pretty good by default, but it's *really* generous with how much space it puts between list items by default. I'm generally not a fan.

Thankfully this is easy to correct with a small snippet:



```

\let\olditem\itemize\renewcommand{\itemize}{\olditem\setlength{\itemsep}{-2ex}}
\let\oldenum\enumerate\renewcommand{\enumerate}{\oldenum\setlength{\itemsep}{-2ex}}
\let\olddesc\description\renewcommand{\description}{\olddesc\setlength{\itemsep}{-2ex}}

```

Then we can just hook this in with our clever preamble.

```

(defvar org-latex-condense-lists t
  "Reduce the space between list items.")
(defvar org-latex-condensed-lists "
<latex-condense-lists>
")

(add-to-list 'org-latex-conditional-features '((and org-latex-condense-lists "^[
\\t]*[+][\\|^\\t]*[1Aa][.]" ) . condensed-lists) t)
(add-to-list 'org-latex-feature-implementations '(condensed-lists :snippet
  ↳ org-latex-condensed-lists :order 0.7) t)

```

## 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.

```

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

```

```

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

```

We'll modify the way listings are generated to make using this as easy as:

```

(setq org-latex-listings 'engraved) ; NOTE non-standard value

```

Thanks to `org-latex-conditional-features` and some copy-paste with the `minted` entry in `org-latex-src-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-src-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-src-block--engraved inline-src-block contents info)
(funcall orig-fn src-block contents info))

(defvar-local org-export-has-code-p nil)

(defadvice! org-export-expect-no-code (&rest _)
  :before #'org-export-as
  (setq org-export-has-code-p nil))

(defadvice! org-export-register-code (&rest _)
  :after #'org-latex-src-block-engraved
  :after #'org-latex-inline-src-block-engraved
  (setq org-export-has-code-p t))

(setq org-latex-engraved-code-preamble "
<org-latex-engraved-code-preamble>
")

(add-to-list 'org-latex-conditional-features '((and org-export-has-code-p "[
  ↳ \t]*#\+begin_src\\|^[\t]*#\+BEGIN_SRC\\|src_[A-Za-z]" . engraved-code) t)
(add-to-list 'org-latex-conditional-features '("[
  ↳ \t]*#\+begin_example\\|^[\t]*#\+BEGIN_EXAMPLE" . engraved-code-setup) t)
(add-to-list 'org-latex-feature-implementations '(engraved-code :requires
  ↳ engraved-code-setup :snippet (engrave-faces-latex-gen-preamble) :order 99) t)
(add-to-list 'org-latex-feature-implementations '(engraved-code-setup :snippet
  ↳ org-latex-engraved-code-preamble :order 98) t)

(defun org-latex-src-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)))
        (float-env (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}"
                caption-str
                caption-str
                caption-str))))

```

```

        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}}

\\definecolor{codebackground}{HTML}{f7f7f7}
\\definecolor{codeborder}{HTML}{f0f0f0}

% TODO have code boxes keep line vertical alignment
\\usepackage[breakable,xparse]{tcolorbox}
\\DeclareTColorBox[]{}{Code}{o}%
{colback=codebackground, colframe=codeborder,
  fontupper=\\footnotesize,
  colupper=EFD,
  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 be 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 compile for code-heavy documents (like this config).

Performing a little benchmark with this document, I find that this is indeed the case.

L <sup>A</sup> T <sub>E</sub> X 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).

## Julia code blocks

Julia code has fantastic support for unicode! The downside is that `pdflatex` is *still* a pain to use with unicode symbols. The solution — `lualatex`. Now we just need to make it automatic

```
(defadvice! org-latex-pick-compiler (_contents info)
  :before #'org-latex-template
  (when (and org-export-has-code-p (memq 'julia-code (org-latex-detect-features)))
    (setf info (plist-put
      (if (member #'org-latex-replace-non-ascii-chars (plist-get info
        ↪ :filter-final-output))
        (plist-put info :filter-final-output
          (delq #'org-latex-replace-non-ascii-chars (plist-get
            ↪ info :filter-final-output)))
        info)
      :latex-compiler "lualatex")))))
```

Then a font with unicode support must be used. JuliaMono is the obvious choice, and we can use it with the `fontspec` package. In future it may be nice to set this just as a fallback font (when it isn't a pain to do so).

```
\usepackage{fontspec}
\newfontfamily\JuliaMono{JuliaMono-Regular.ttf}[Path=/usr/share/fonts/truetype/,
↪ Extension=.ttf]
\newfontface\JuliaMonoRegular{JuliaMono-Regular}
\setmonofont{JuliaMonoRegular}[Contextuals=Alternate, Scale=MatchLowercase]
```

Now all that remains is to hook this into the preamble generation.

```
(setq org-latex-julia-mono-fontspec "
<<julia-mono-fontspec>>
")

(add-to-list 'org-latex-feature-implementations '(julia-code :snippet
↪ org-latex-julia-mono-fontspec :order 0) t)
(add-to-list 'org-latex-conditional-features '((and org-export-has-code-p "[
↪ \t]*#\+begin_src julia\\|^[\t]*#\+BEGIN_SRC julia\\|src_julia") . julia-code)
↪ t)

(add-to-list 'org-latex-feature-implementations '(.microtype-lualatex :eager t :when
↪ (microtype julia-code) :prevents microtype :order 0.1 :snippet
↪ "\\usepackage[activate={true,nocompatibility},final,tracking=true,factor=2000]{microtype}\n"))
(add-to-list 'org-latex-feature-implementations '(.custom-font-no-mono :eager t
↪ :prevents custom-font :order 0 :snippet (org-latex-fontset :serif :sans)) t)
```

## Emojis

It would be nice to actually include emojis where used. Thanks to `emojify`, we have a folder of emoji images just sitting and waiting to be used.

First up, we want to detect when emojis are actually present. We can try checking the unicode ranges with a collection of `[?~?]` regex groups, but Emojis are actually spread around a fair bit and so this isn't very straightforward. Instead I can iterate thorough non-ASCII characters and check if any have the text property `emojiified`.

```
(defun emojify-emoji-in-buffer-p ()
  "Determine if any emojis are present in the current buffer, using `emojify-mode'."
  (unless emojify-mode
    (emojify-mode 1)
    (emojify-display-emojis-in-region (point-min) (point-max)))
  (let (emoji-found end)
    (save-excursion
      (goto-char (point-min))
      (while (not (or emoji-found end))
        (if-let ((pos (re-search-forward "[^[:ascii:]]" nil t)))
          (when (get-text-property (1- pos) 'emojiified)
            (setq emoji-found t))
          (setq end t))))
    emoji-found))
```

Once we know that there are emojis present we can add a bit of preamble to the buffer to make insertion easier.

```
(defun org-latex-emoji-setup ()
  (format
    ↪ "\newcommand\emoji[1]{\raisebox{-0.3ex}{\includegraphics[height=1.8ex]{%s/#1}}}"
    ↪ (emojify-image-dir))

  (add-to-list 'org-latex-conditional-features '((emojify-emoji-in-buffer-p) . emoji) t)
  (add-to-list 'org-latex-feature-implementations '(emoji :requires image :snippet
    ↪ (org-latex-emoji-setup) :order 3 ))
```

Once again making use of `emojify`, we can generate  $\LaTeX$  commands for our emojis fairly easily.

```
(defun emojify-latexify-emoji-in-buffer ()
  (unless emojify-mode
    (emojify-mode 1)
    (emojify-display-emojis-in-region (point-min) (point-max)))
  (let (end)
    (save-excursion
      (goto-char (point-min))
      (while (not end)
```

```
(if-let ((pos (re-search-forward "[^[:ascii:]]\\{1,2\\}" nil t)))
  (when-let ((char (get-text-property (1- pos) 'emojify-text))
             (emoji (emojify-get-emoji char)))
    (replace-match (format "\\ \\emoji{%s}" (file-name-sans-extension (ht-get
      ↪ emoji "image")))))
  (setq end t))))
```

Now we just need to hook this handy function into Org's export. We can't use standard string-replacement as we rely on the buffer modifications enacted by (emojify-mode).

As I have not yet implemented a nice way of sharing feature detection information outside of (org-latex-generate-features-preamble), we'll use the same check before attempting to LaTeXify emojis and hope that nothing strange happens.

```
(defun +org-latex-convert-emojis (text backend _info)
  (when (org-export-derived-backend-p backend 'latex)
    (with-temp-buffer
      (insert text)
      (when (emojify-emoji-in-buffer-p)
        (emojify-latexify-emoji-in-buffer)
        (buffer-string))))
  (add-to-list 'org-export-filter-final-output-functions #' +org-latex-convert-emojis))
```

This works fairly nicely, there's just one little QOL upgrade that we can perform. emojify downloads the 72x72 versions of Twemoji, however SVG versions are also produced. We could use inkscape to convert those to PDFs, which would likely be best for including.

First, it's worth checking whether .pdf graphics files will be prioritised over .png files. If so, that would be ideal as no extra effort is required past fetching and converting the files.

```
texdef -t pdflatex -p graphicx Gin@extensions
```

```
\Gin@extensions:
macro:->.pdf,.png,.jpg,.mps,.jpeg,.jbig2,.jb2,.PDF,.PNG,.JPG,.JPEG,.JBIG2,.JB2,.eps
```

Fantastic! We can see that .pdf actually comes first in the priority list. So now we just need to fetch and convert those SVGs — ideally with a handy command to do so for us.

```
(defun org-latex-emoji-install-vector-graphics ()
  "Download, convert, and install vector emojis for use with LaTeX."
  (interactive)
  (let ((dir (org-latex-emoji-install-vector-graphics--download)))
    (org-latex-emoji-install-vector-graphics--convert dir)
    (org-latex-emoji-install-vector-graphics--install dir)
    (message "Vector emojis installed."))

(defun org-latex-emoji-install-vector-graphics--download ()
```



```

(message "Locating latest emojis...")
(let* ((twemoji-url (substring (shell-command-to-string "echo
↳ \"https://github.com$(curl -sL
↳ https://github.com/twitter/twemoji/releases/latest | grep '.zip\"' | cut -d '\"'
↳ -f 2)\"") 0 -1))
      (twemoji-version (replace-regexp-in-string "^.*tags/v\\(.*\\)\\.zip" "\\1"
↳ twemoji-url))
      (twemoji-dest-folder (make-temp-file "twemoji-" t)))
(message "Downloading Twemoji v%s" twemoji-version)
(let ((default-directory twemoji-dest-folder))
  (call-process "curl" nil nil nil "-L" twemoji-url "--output" "twemoji.zip")
  (message "Unzipping")
  (call-process "unzip" nil nil nil "twemoji.zip")
  (concat twemoji-dest-folder "/twemoji-" twemoji-version "/assets/svg"))))

(defun org-latex-emoji-install-vector-graphics--convert (dir)
  (let ((default-directory dir))
    (if (executable-find "cairosvg") ; cairo's PDFs are ~10% smaller
        (let* ((images (directory-files dir nil ".*.svg"))
              (num-images (length images))
              (index 0)
              (max-threads (1- (string-to-number (shell-command-to-string "nproc"))))
              (threads 0))
          (while (< index num-images)
            (setf threads (1+ threads))
            (message "Converting emoji %d/%d (%s)" (1+ index) num-images (nth index
↳ images))
            (make-process :name "cairosvg"
                          :command (list "cairosvg" (nth index images) "-o" (concat
↳ (file-name-sans-extension (nth index images)) ".pdf"))
                          :sentinel (lambda (proc msg)
                                      (when (memq (process-status proc) '(exit
↳ signal))
                                        (setf threads (1- threads)))))
            (setf index (1+ index))
            (while (> threads max-threads)
              (sleep-for 0.01)))
          (while (> threads 0)
            (sleep-for 0.01))
          (message "Finished conversion!"))
        (shell-command "inkscape --batch-process --export-type='pdf' *.svg"))))

(defun org-latex-emoji-install-vector-graphics--install (dir)
  (message "Installing vector emojis into emoji directory")
  (let ((images (directory-files dir t ".*.pdf"))
        (emoji-dir (concat (emojify-image-dir) "/")))
    (mapcar
     (lambda (image)

```

```
(rename-file image emoji-dir t))
images)))
```

## 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.

We check every non-ascii character to make sure it's not a character encoded by the `inputenc` packages when loaded with the `utf8` option. Finally, we see if we have our own `LATEX` conversion we can apply and if there is none we replace the non-ascii char with  $\zeta$ .

No to make sure we only remove characters that can't be displayed, we check `/usr/share/texmf/tex/latex/base/`

We just need to make sure this is appended to the list of filter functions, since we want to let emoji processing occur first.

```
(defvar +org-pdflatex-inputenc-encoded-chars

  ↳ "[[:ascii:]]\u00A0-\u01F0\u0218-\u021Bÿj^~`~\u0400-\u04FFßŠ\u200C\u2010-\u201E†•…%‰‹›※?/*%‰€£¥

(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")))
    (let (case-replace)
      (replace-regexp-in-string "[^[:ascii:]]"
                                (lambda (nonascii)
                                  (if (string-match-p
                                        ↳ +org-pdflatex-inputenc-encoded-chars nonascii)
                                      ↳ nonascii
                                      (or (cdr (assoc nonascii
                                                         ↳ +org-latex-non-ascii-char-substitutions))
                                          ↳ "¿"))))
      text))))

(add-to-list 'org-export-filter-final-output-functions
  ↳ #' +org-latex-replace-non-ascii-chars t)
```

Now, there are some symbols that aren't included in `inputenc`, but we should be able to handle anyway. For them we define a table of `LATEX` translations

Character	L <sup>A</sup> T <sub>E</sub> X
$\alpha$	$\alpha$
$\beta$	$\beta$
$\gamma$	$\gamma$
$\delta$	$\delta$
$\epsilon$	$\epsilon$
$\varepsilon$	$\varepsilon$
$\zeta$	$\zeta$
$\eta$	$\eta$
$\theta$	$\theta$
$\vartheta$	$\vartheta$
$\iota$	$\iota$
$\kappa$	$\kappa$
$\lambda$	$\lambda$
	$\mu$
$\nu$	$\nu$
$\xi$	$\xi$
$\pi$	$\pi$
$\omega$	$\omega$
$\rho$	$\rho$
$\varrho$	$\varrho$
$\sigma$	$\sigma$
$\varsigma$	$\varsigma$
$\tau$	$\tau$
$\upsilon$	$\upsilon$
$\phi$	$\phi$
$\varphi$	$\varphi$
$\psi$	$\psi$
	$\omega$
	$\Gamma$
	$\Delta$
	$\Theta$
	$\Lambda$
	$\Xi$
	$\Pi$
	$\Sigma$
	$\Upsilon$
	$\Phi$
	$\Psi$
	$\Omega$
$\aleph$	$\aleph$
$\beth$	$\beth$
$\lrcorner$	$\lrcorner$
$\daleth$	$\daleth$

```
(replace-regexp-in-string
  " '(" "\n  '("
  (replace-regexp-in-string
    ") (" ") \n    ("
    (prin1-to-string
      `(defvar +org-latex-non-ascii-char-substitutions
        ',(mapcar
          (lambda (entry)
            (cons (car entry) (replace-regexp-in-string "\\\\" "\\\\\\\\\\\\\\\\") (cadr
              ↪ entry))))
          latex-non-ascii-char-substitutions))))))
```

```
<<gen-latex-non-ascii-char-substitutions(>>
```

## Normal spaces after abbreviations

In  $\text{\LaTeX}$  inter-word and sentence spaces are typically of different widths. This can be an issue when using abbreviations i.e. e.g. etc. et al.. This can be corrected by forcing a normal space with `.` When exporting Org documents, we can add a filter to check for common abbreviations and make the space normal.

```
(defvar +org-latex-abbreviations
  ';; Latin
    "cf." "e.g." "etc." "et al." "i.e." "v." "vs." "viz." "n.b."
  ;; Corporate
    "inc." "govt." "ltd." "pty." "dept."
  ;; Temporal
    "est." "c."
  ;; Honorifics
    "Prof." "Dr." "Mr." "Mrs." "Ms." "Miss." "Sr." "Jr."
  ;; Components of a work
    "ed." "vol." "sec." "chap." "pt." "pp." "op." "no."
  ;; Common usage
    "approx." "misc." "min." "max.")
  "A list of abbreviations that should be spaced correctly when exporting to LaTeX.")

(defun +org-latex-correct-latin-abbreviation-spaces (text backend _info)
  "Normalise spaces after Latin abbreviations."
  (when (org-export-derived-backend-p backend 'latex)
    (replace-regexp-in-string (rx (group (or line-start space)
      (regexp (regexp-opt-group
        ↪ +org-latex-abbreviations)))
      (or line-end space))
      "\\1\\\\ "
      text)))
```

```
(add-to-list 'org-export-filter-paragraph-functions
  ↪ #'org-latex-correct-latin-abbreviation-spaces t)
```

## Extra special strings

`\TeX` already recognises `---` and `--` as em/en-dashes, `\-` as a shy hyphen, and the conversion of `...` to `\ldots{}` is hardcoded into `org-latex-plain-text` (unlike `org-html-plain-text`).

I'd quite like to also recognise `->` and `<-`, so let's set come up with some advice.

```
(defvar org-latex-extra-special-string-regexps
  '((->" . "\\|\\|\\|\\textrightarrow{}")
    (<-;" . "\\|\\|\\|\\textleftarrow{"})))

(defun org-latex-convert-extra-special-strings (string)
  "Convert special characters in STRING to LaTeX."
  (dolist (a org-latex-extra-special-string-regexps string)
    (let ((re (car a))
          (rpl (cdr a)))
      (setq string (replace-regexp-in-string re rpl string t)))))

(defadvice! org-latex-plain-text-extra-special-a (orig-fn text info)
  "Make `org-latex-plain-text' handle some extra special strings."
  :around #'org-latex-plain-text
  (let ((output (funcall orig-fn text info)))
    (when (plist-get info :with-special-strings)
      (setq output (org-latex-convert-extra-special-strings output)))
    output))
```

## Support images from URLs

You can link to remote images easily, and they work nicely with HTML-based exports. However,  $\text{\LaTeX}$  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 I had the idea of having the look of the  $\text{\LaTeX}$  document produced match the current Emacs theme, I was enraptured. The result is the pseudo-class `chameleon`, which I have implemented in the package `ox-chameleon`.

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

```
(use-package! ox-chameleon
  :after ox)
```

### 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 . "\\uuline{%s}")
        (verbatim . verb)))
```

### Check for required packages

For how I've setup Org's  $\text{\LaTeX}$  export, the following packages are needed:

- `adjustbox`
- `arev`
- `amsmath`
- `booktabs`
- `cancel`
- `capt-of`
- `caption`
- `cleveref`
- `embedall`
- `fourier`
- `fvextra`
- `gillius`
- `graphicx`
- `hyperref`
- `mathalpha`
- `mathtools`
- `microtype`
- `pdfx`

- pifont
- preview
- siunitx
- soul
- subcaption
- svg
- tcolorbox
- xcolor
- xparse

Then for the various fontsets:

- Alegreya
- biolinum
- FiraMono
- FiraSans
- kpfonts
- libertine
- newpxmath
- newpxtext
- newtxmath
- newtxtext
- newtxsf
- noto
- plex-mono
- plex-sans
- plex-serif
- sourcecodepro
- sourcesanspro
- sourceserifpro

We can write a function which will check for each of these packages with `kpsewhich`, and then if any of them are missing we'll inject some advice into the generated config that gets a list of missing packages and warns us every time we export to a PDF.



```

(setq org-required-latex-packages (append (mapcar #'car
  ↪ org-latex-required-packages-list)
                                         (mapcar #'car
  ↪ org-latex-font-packages-list)))

(defun check-for-latex-packages (packages)
  (delq nil (mapcar (lambda (package)
    (unless
      (= 0 (shell-command (format "kpsewhich %s.sty" package)))
      package))
    packages)))

(if-let ((missing-pkgs (check-for-latex-packages org-required-latex-packages)))
  (concat
    (pp-to-string `(setq org-required-latex-packages ',org-required-latex-packages))
    (message ";; Detected missing LaTeX packages: %s\n" (mapconcat #'identity
  ↪ missing-pkgs ", ")))
  (pp-to-string
    '(defun check-for-latex-packages (packages)
      (delq nil (mapcar (lambda (package)
        (unless
          (= 0 (shell-command (format "kpsewhich %s.sty"
  ↪ package)))
          package))
        packages))))

  (pp-to-string
    '(defun +org-warn-about-missing-latex-packages (&rest _)
      (message "Checking for missing LaTeX packages...")
      (sleep-for 0.4)
      (if-let ((missing-pkgs (check-for-latex-packages org-required-latex-packages)))
        (message "%s You are missing the following LaTeX packages: %s."
          (propertyize "Warning!" 'face '(bold warning))
          (mapconcat (lambda (pkg) (propertyize pkg 'face
  ↪ 'font-lock-variable-name-face))
            missing-pkgs
            ", "))
        (message "%s You have all the required LaTeX packages. Run %s to make this
  ↪ message go away."
          (propertyize "Success!" 'face '(bold success))
          (propertyize "doom sync" 'face 'font-lock-keyword-face))
        (advice-remove 'org-latex-export-to-pdf
  ↪ #' +org-warn-about-missing-latex-packages))
      (sleep-for 1)))

  (pp-to-string
    '(advice-add 'org-latex-export-to-pdf :before
  ↪ #' +org-warn-about-missing-latex-packages)))
  ";; No missing LaTeX packages detected")

```

```
<<org-missing-latex-packages(>>
```

### 5.3.8 Beamer Export

It's nice to use a different theme

```
(setq org-beamer-theme "[progressbar=foot]metropolis")
```

```
(defun org-beamer-p (info)
  (eq 'beamer (and (plist-get info :back-end) (org-export-backend-name (plist-get info
    ↪ :back-end))))))

(add-to-list 'org-latex-conditional-features '(org-beamer-p . beamer) t)
(add-to-list 'org-latex-feature-implementations '(beamer :requires .missing-koma
  ↪ :prevents (italic-quotes condensed-lists)) t)
(add-to-list 'org-latex-feature-implementations '(.missing-koma :snippet
  ↪ "\\usepackage{scrextend}" :order 2) t)
```

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)
```

### 5.3.9 Reveal export

By default reveal is rather nice, there are just a few tweaks that I consider a good idea.

```
(setq org-re-reveal-theme "white"
      org-re-reveal-transition "slide"
      org-re-reveal-plugins '(markdown notes math search zoom))
```

### 5.3.10 ASCII export

To start with, why settle for ASCII when UTF-8 exists?

```
(setq org-ascii-charset 'utf-8)
```

The ASCII export is generally fairly nice. I think the main aspect that could benefit from improvement is the appearance of  $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$  fragments. There's a nice utility we can use to create unicode representation, which are much nicer. It's called `latex2text`, and it's part of the `pylatexenc` package, and it's [not really packaged](#). So, we'll resort to installing it with `pip`.

```
sudo python3 -m pip install pylatexenc
```

With that installed, we can override the `(org-ascii-latex-fragment)` and `(org-ascii-latex-environment)` functions, which are conveniently very slim — just extracting the content, and indenting. We'll only do something different when `utf-8` is set.

```
(after! ox-ascii
  (defvar org-ascii-convert-latex t
    "Use latex2text to convert LaTeX elements to unicode.")

  (defadvice! org-ascii-latex-environment-unicode-a (latex-environment _contents info)
    "Transcode a LATEX-ENVIRONMENT element from Org to ASCII, converting to unicode.
     CONTENTS is nil. INFO is a plist holding contextual
     information."
    :override #'org-ascii-latex-environment
    (when (plist-get info :with-latex)
      (org-ascii--justify-element
       (org-remove-indentation
        (let* ((latex (org-element-property :value latex-environment))
              (unicode (and (eq (plist-get info :ascii-charset) 'utf-8)
                            org-ascii-convert-latex
                            (doom-call-process "latex2text" "-q" "--code" latex))))
          (if (= (car unicode) 0) ; utf-8 set, and successfully ran latex2text
              (cdr unicode) latex)))
        latex-environment info)))

  (defadvice! org-ascii-latex-fragment-unicode-a (latex-fragment _contents info)
    "Transcode a LATEX-FRAGMENT object from Org to ASCII, converting to unicode.
     CONTENTS is nil. INFO is a plist holding contextual
     information."
    :override #'org-ascii-latex-fragment
    (when (plist-get info :with-latex)
      (let* ((latex (org-element-property :value latex-fragment))
            (unicode (and (eq (plist-get info :ascii-charset) 'utf-8)
                          org-ascii-convert-latex
                          (doom-call-process "latex2text" "-q" "--code" latex))))
        (if (and unicode (= (car unicode) 0)) ; utf-8 set, and successfully ran
            ↪ latex2text
            (cdr unicode) latex))))))
```

### 5.3.11 Markdown Export

**GFM**

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 ox)
```

### Character substitutions

When I want to paste exported markdown somewhere (for example when using Emacs Everywhere), 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.

Another worthwhile consideration is  $\text{\LaTeX}$  formatting. It seems most Markdown parsers are fixated on  $\text{\TeX}$ -style syntax ( $\$$  and  $\$\$$ ). As unfortunate as this is, it's probably best to accommodate them, for the sake of decent rendering.

ox-md doesn't provide any transcoders for this, so we'll have to whip up our own and push them onto the md transcoders alist.

```

(after! ox-md
  (defun org-md-latex-fragment (latex-fragment _contents info)
    "Transcode a LATEX-FRAGMENT object from Org to Markdown."
    (let ((frag (org-element-property :value latex-fragment)))
      (cond
        ((string-match-p "\\\" frag)
         (concat "$" (substring frag 2 -2) "$"))
        ((string-match-p "\\\" frag)
         (concat "$$" (substring frag 2 -2) "$$"))
        (t (message "unrecognised fragment: %s" frag)
            frag))))

  (defun org-md-latex-environment (latex-environment contents info)
    "Transcode a LATEX-ENVIRONMENT object from Org to Markdown."
    (concat "$$\n"
            (org-html-latex-environment latex-environment contents info)
            "$$\n"))

  (defun org-utf8-entity (entity _contents _info)
    "Transcode an ENTITY object from Org to utf-8.
     CONTENTS are the definition itself. INFO is a plist holding
     contextual information."
    (org-element-property :utf-8 entity))

;; We can't let this be immediately parsed and evaluated,
;; because eager macro-expansion tries to call as-of-yet
;; undefined functions.
;; NOTE in the near future this shouldn't be required
(eval
  '(dolist (extra-transcoder
            '((latex-fragment . org-md-latex-fragment)
              (latex-environment . org-md-latex-environment)
              (entity . org-utf8-entity)))
    (unless (member extra-transcoder (org-export-backend-transcoders
                                      (org-export-get-backend 'md)))
      (push extra-transcoder (org-export-backend-transcoders
                              (org-export-get-backend 'md)))))))

```

### 5.3.12 Babel

Doom lazy-loads babel languages, with is lovely. It also pulls in [ob-async](#), which is nice, but it would be even better if it was used by default.

There are two caveats to ob-async:

1. It does not support `:session`
  - So, we don't want `:async` used when `:session` is set
2. It adds a fixed delay to execution
  - This is undesirable in a number of cases, for example it's generally unwanted with `emacs-lisp` code
  - As such, I also introduce a `async` language blacklist to control when it's automatically enabled

Due to the nuance in the desired behaviour, instead of just adding `:async` to `org-babel-default-header-args`, I advice `org-babel-get-src-block-info` to add `:async` intelligently. As an escape hatch, it also recognises `:sync` as an indication that `:async` should not be added.

I did originally have this enabled for everything except for `emacs-lisp` and `LaTeX` (there were weird issues), but this added a ~3s “startup” cost to every `src` block evaluation, which was a bit of a pain. Since `:async` can be added easily with `#+properties`, I've turned this behaviour from a blacklist to a whitelist.

```
(add-transient-hook! #'org-babel-execute-src-block
  (require 'ob-async))

(defvar org-babel-auto-async-languages '()
  "Babel languages which should be executed asynchronously by default.")

(defadvice! org-babel-get-src-block-info-eager-async-a (orig-fn &optional light datum)
  "Eagerly add an :async parameter to the src information, unless it seems
   problematic.
   This only acts o languages in `org-babel-auto-async-languages'.
   Not added when either:
   + session is not \"none\"
   + :sync is set"
  :around #'org-babel-get-src-block-info
  (let ((result (funcall orig-fn light datum)))
    (when (and (string= "none" (cdr (assoc :session (caddr result))))
              (member (car result) org-babel-auto-async-languages)
              (not (assoc :async (caddr result))) ; don't duplicate
              (not (assoc :sync (caddr result))))
      (push '(:async) (caddr result)))
    result))
```

### 5.3.13 ESS

We don't want R evaluation to hang the editor, hence

```
(setq ess-eval-visibly 'nowait)
```

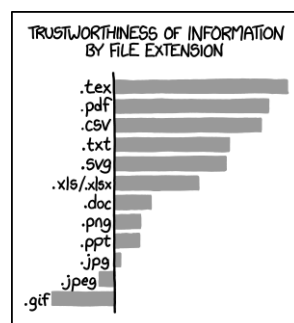
Syntax highlighting is nice, so let's turn all of that on

```
(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)))
```

Lastly, in the event that I use JAGS, it would be nice to be able to use jags as the language identifier, not ess-jags.

```
(add-to-list '+org-babel-mode-alist '(jags . ess-jags))
```

## 5.4 L<sup>A</sup>T<sub>E</sub>X



**File Extensions** I have never been lied to by data in a .txt file which has been hand-aligned.

### 5.4.1 To-be-implemented ideas

- Paste image from clipboard
  - Determine first folder in graphicspath if applicable

- 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 `yasnipet` with filename as variable maybe?)

## 5.4.2 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))
```

## 5.4.3 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 (completing-read "Select document class: "
    ↪ '("article" "scrartcl" "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)))
```

## Delimiters

```
(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 character (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 delimiter,
    ↪ recognises: 9 ( [ { < | .")
    (when (eql ?. tec/tex-last-delim-char)
      (setq tec/tex-delim-dot-second (read-char-exclusive "Other delimiter,
    ↪ 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  "\\9")
  (?\\[ "\\[")
  (?\\{ "\\{\\}")
  (?<  "\\<")
  (?\\) "\\)")
  (?0  "\\0")
  (?\\] "\\]")
  (?\\} "\\}\\}")
  (?\\> "\\>")
  (?|  "\\|")
  (_   "\\.")))))
(defun tec/tex-next-char-smart-close-delim (&optional open-char)
  (and (bound-and-true-p smartparens-mode)
    (eq (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 "_delimiters" 'latex-mode) (point) (+
    ↪ (point) (if (tec/tex-next-char-smart-close-delim open-char) 2 1))))

```

## 5.4.4 Editor visuals

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"))
    ;; convenience shortcuts -- these don't work nicely ATM
    ; ("<" ("left"))

```

```

;; (>" ("right"))
;; private macros
("z" ("RR"))
("z" ("NN"))
("z" ("ZZ"))
("z" ("QQ"))
("z" ("CC"))
("z" ("PP"))
("z" ("HH"))
("z" ("EE"))
("z" ("dd"))
;; known commands
(" " ("phantom"))
(, (lambda (num den) (if (and (TeX-string-single-token-p num)
    ↪ (TeX-string-single-token-p den))
        (concat num "z" den)
        (concat "z" num "z" den "z"))) ("frac"))
(, (lambda (arg) (concat "z" (TeX-fold-parenthesize-as-necessary arg)))
    ↪ ("sqrt"))
(, (lambda (arg) (concat "z" (TeX-fold-parenthesize-as-necessary arg)))
    ↪ ("vec"))
("{'1}' ("text"))
;; private commands
("|{1}|" ("abs"))
("||{1}||" ("norm"))
("z{1}z" ("floor"))
("z{1}z" ("ceil"))
("z{1}z" ("round"))
("z{1}/z{2}" ("dv"))
("z{1}/z{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 character 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) ; i
    ;; lowercase caligraphic
    (119994 . 8495) ; i
    (119996 . 8458) ; i
    (120004 . 8500) ; i
    ;; caligraphic
    (119965 . 8492) ; i
    (119968 . 8496) ; i
    (119969 . 8497) ; i
    (119971 . 8459) ; i
    (119972 . 8464) ; i
    (119975 . 8466) ; i
    (119976 . 8499) ; i
    (119981 . 8475) ; i
    ;; fraktur
    (120070 . 8493) ; i
    (120075 . 8460) ; i
    (120076 . 8465) ; i

```

```

(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-necessary (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 delimiters can be de-emphasised a bit

```

;; Making \ ( \) less visible
(defface unimportant-latex-face
  '((t :inherit font-lock-comment-face :weight extra-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 "\\}"))
```

## 5.4.5 Math input

### 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-env-alist
 '(("bmatrix" "\\begin{bmatrix}\\n?\\n\\end{bmatrix}" nil)
 ("equation*" "\\begin{equation*}\\n?\\n\\end{equation*}" nil)))
 (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 convenience
 (? . ("\\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)))
```

## LAAS

This makes use of `aas` (*Auto Activating Snippets*) for C<sub>D</sub>LaTeX-like symbol input.

```
(package! laas :recipe (:local-repo "lisp/LaTeX-auto-activating-snippets"))
```

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

### 5.4.6 SyncTeX

```
(after! tex
  (add-to-list 'TeX-view-program-list '("Evince" "evince %o"))
  (add-to-list 'TeX-view-program-selection '(output-pdf "Evince")))
```

### 5.4.7 Fixes

In case of Emacs28,

```
(when EMACS28+
  (add-hook 'latex-mode-hook #'TeX-latex-mode))
```

## 5.5 Python

Since I'm using `mypy-ls`, as suggested in [:lang python LSP support](#) I'll tweak the priority of `mypy-ls`

```
(after! lsp-python-ms
  (set-lsp-priority! 'mspyls 1))
```

## 5.6 PDF

### 5.6.1 MuPDF

pdf-tools is nice, but a mupdf-based solution is nicer.

```
(package! paper :recipe (:host github :repo "ymarco/paper-mode"
                          :files ("*.el" ".so")
                          :pre-build ("make")))
```

```
;; (use-package paper
;;   ;; :mode ("\\.pdf\\'" . paper-mode)
;;   ;; :mode ("\\.epub\\'" . paper-mode)
;;   :config
;;   (require 'evil-collection-paper)
;;   (evil-collection-paper-setup))
```

### 5.6.2 Terminal viewing

Sometimes I'm in a terminal and I still want to see the content. Additionally, sometimes I'd like to act on the textual content and so would like a plaintext version. Thanks to we have a convenient way of performing this conversion. I've integrated this into a little package, `pdftotext.el`.

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

The output can be slightly nicer without spelling errors, and with prettier page feeds (^L by default).

This is very nice, now we just need to associate it with `.pdf` files, and make sure `pdf-tools` doesn't take priority.

Lastly, whenever Emacs is non-graphical (i.e. a TUI), we want to use this by default.

```
(use-package! pdftotext
  :init
  (unless (display-graphic-p)
```



```

(add-to-list 'auto-mode-alist '("\\\\. [pP][dD][fF]\\\\" . pdftotext-mode))
(add-to-list 'magic-mode-alist '("%PDF" . pdftotext-mode))
:config
(unless (display-graphic-p) (after! pdf-tools (pdftotext-install)))
;; For prettyness
(add-hook 'pdftotext-mode-hook #'spell-fu-mode-disable)
(add-hook 'pdftotext-mode-hook (lambda () (page-break-lines-mode 1)))
;; I have no idea why this is needed
(map! :map pdftotext-mode-map
      "<mouse-4>" (cmd! (scroll-down mouse-wheel-scroll-amount-horizontal))
      "<mouse-5>" (cmd! (scroll-up mouse-wheel-scroll-amount-horizontal)))

```

## 5.7 R

### 5.7.1 Editor Visuals

```

(after! ess-r-mode
  (append! +ligatures-extra-symbols
    '(:assign "¿"
      :multiply "x"))
  (set-ligatures! 'ess-r-mode
    ;; Functional
    :def "function"
    ;; Types
    :null "NULL"
    :true "TRUE"
    :false "FALSE"
    :int "int"
    :float "float"
    :bool "bool"
    ;; Flow
    :not "!"
    :and "&&" :or "||"
    :for "for"
    :in "%in%"
    :return "return"
    ;; Other
    :assign "<-"
    :multiply "%*%"))

```

## 5.8 Julia

As mentioned in [lsp-julia#35](#), lsp-mode seems to serve an invalid response to the Julia server. The pseudo-fix is rather simple at least

```
(after! julia-mode
  (add-hook 'julia-mode-hook #'rainbow-delimiters-mode-enable)
  (add-hook! 'julia-mode-hook
    (setq-local lsp-enable-folding t
      lsp-folding-range-limit 100)))
```

## 5.9 Graphviz

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

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

(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)
```

## 5.10 Markdown

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))
```

## 5.11 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-compatible (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.

```
(package! beancount :recipe (:host github :repo "beancount/beancount-mode")
  :pin "ea8257881b7e276e8d170d724e3b2e179f25cb77")

(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)))
```

## 5.12 Snippets

### 5.12.1 Latex mode

#### File template

```
# -*- mode: snippet -*-
# name: LaTeX template
↵
# --
\documentclass${1:[${2:opt1,...}]}{(tec/yas-latex-get-class-choice)}`}

\title{${3:~(s-titleized-words (file-name-base (or buffer-file-name "new buffer")))}~}`}
\author{${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}
```

#### Delimiters

```
# name: _delimiters
# --
\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}
```

## Delimiters

```
# -*- mode: snippet -*-
# name: delimiters
# key: @
# condition: (texmathp)
# type: command
# --
(tec/get-open-delim-char)
(yas-expand-snippet (yas-lookup-snippet "_delimiters" 'latex-mode))
```

## Delimiters angle

```
# -*- mode: snippet -*-
# name: delimiters - angle <>
# key: <
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\<)
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

## Delimiters bracket

```
# -*- mode: snippet -*-
# name: delimiters - bracket []
# key: [
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\[)
(setq tec/tex-delim-dot-second nil)
(tec/tex-delim-yas-expand)
```

## Delimiters curly

```
# -*- mode: snippet -*-
# name: delimiters - curly {}
# key: {
# condition: (texmathp)
# type: command
# --
(setq tec/tex-last-delim-char ?\{)
(setq tec/tex-delim-dot-second nil)
```

```
(tec/tex-delim-yas-expand)
```

## Delimiters paren

```
# -*- mode: snippet -*-
# name: delimiters - 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}
```

### 5.12.2 Markdown mode

#### File template

```
# -*- mode: snippet -*-
# name: Org template
# --
# ${1:~(s-titleized-words (file-name-base (or buffer-file-name "new buffer")))}
$0
```

### 5.12.3 Org mode

#### File template

```
# -*- mode: snippet -*-
# name: Org template
# --
#+title: ${1:~(s-titleized-words (replace-regexp-in-string
↳ "[0-9]\\{4\\}-[0-9][0-9]-[0-9][0-9]-" "" (file-name-base (or buffer-file-name
↳ "new buffer"))))}`}
#+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-latex-fragment)
↳ (evil-insert-state) (insert "\n \n") (left-char))))
```

```
# --
\\[`%`$0\\]
```

### Elisp source

```
# -*- 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 argument 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 argument eval

```
# -*- mode: snippet -*-
# name: Header arg - eval
# key: v
# condition: (+yas/org-src-header-p)
# --
`(let ((out (+yas/org-prompt-header-arg :eval "Evaluate: " '("no" "query" "no-export"
↳ "query-export")))) (if out (concat ":eval " out " ") ""))`
```

### Header argument export

```
# -*- mode: snippet -*-
# name: Header arg - export
# key: e
```

```
# condition: (+yas/org-src-header-p)
# --
`(let ((out (+yas/org-prompt-header-arg :exports "Exports: " '("code" "results" "both"
↳ "none")))) (if out (concat ":exports " out " ") ""))`
```

### Header argument file

```
# -*- mode: snippet -*-
# name: Header arg - file
# key: f
# condition: (+yas/org-src-header-p)
# --
:file $0
```

### Header argument graphics

```
# -*- mode: snippet -*-
# name: Header arg - graphics
# key: g
# condition: (+yas/org-src-header-p)
# --
:results file graphics $0
```

### Header argument height

```
# -*- mode: snippet -*-
# name: Header arg - height
# key: H
# condition: (+yas/org-src-header-p)
# --
:height $0
```

### Header argument noweb

```
# -*- mode: snippet -*-
# name: Header arg - noweb
# key: n
# condition: (+yas/org-src-header-p)
# --
`(let ((out (+yas/org-prompt-header-arg :noweb "NoWeb: " '("no" "yes" "tangle"
↳ "no-export" "strip-export" "eval")))) (if out (concat ":noweb " out " ") ""))`
```

### Header argument output

```
# -*- mode: snippet -*-
# name: Header arg - output
# key: o
# condition: (+yas/org-src-header-p)
# --
```

```
:results output $0
```

### Header argument results

```
# -*- mode: snippet -*-
# name: Header arg - results
# key: r
# condition: (+yas/org-src-header-p)
# --
~(let ((out
(string-trim-right
(concat
(+yas/org-prompt-header-arg :results "Result collection: " ("value " "output "))
(+yas/org-prompt-header-arg :results "Results type: " ("table " "vector " "list "
↪ "verbatim " "file "))
(+yas/org-prompt-header-arg :results "Results format: " ("code " "drawer " "html "
↪ "latex " "link " "graphics " "org " "pp " "raw "))
(+yas/org-prompt-header-arg :results "Result output: " ("silent " "replace "
↪ "append " "prepend "))))))
(if (string= out "") ""
(concat ":results " out " ")))
~
```

### Header argument session

```
# -*- mode: snippet -*-
# name: Header arg - session
# key: S
# condition: (+yas/org-src-header-p)
# --
:session "${1:~(file-name-base (or (buffer-file-name) "unnamed"))~-session}" $0
```

### Header argument silent

```
# -*- mode: snippet -*-
# name: Header arg - silent
# key: s
# condition: (+yas/org-src-header-p)
# --
:results silent $0
```

### Header argument tangle

```
# -*- mode: snippet -*-
# name: Header arg - tangle
# key: t
# condition: (+yas/org-src-header-p)
# --
:tangle $0
```

## Header argument width

```
# -*- mode: snippet -*-
# name: Header arg - width
# key: W
# condition: (+yas/org-src-header-p)
# --
:width $0
```

## Header argument wrap

```
# -*- mode: snippet -*-
# name: Header arg - wrap
# key: w
# condition: (+yas/org-src-header-p)
# --
`(let ((out (+yas/org-prompt-header-arg :noweb "Wrap: " '("example" "export" "comment"
↳ "src")))) (if out (concat ":wrap " out " ") ""))`
```

## Inline math

```
# -*- mode: snippet -*-
# name: inline math
# key: m
# condition: t
# expand-env: ((yas-after-exit-snippet-hook (lambda () (org-edit-latex-fragment)
↳ (evil-insert-state) (goto-char 3))))
# --
\\(`%`$0\\)
```

## Property header arguments

```
# -*- 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 source

```
# -*- 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
```

### Source

```
# -*- mode: snippet -*-
# name: #+begin_src
# uuid: src
# key: src
# --
#+begin_src ${1: `(or (+yas/org-last-src-lang) "?")` }
`%`$0
#+end_src
```