My Emacs Initialisation File, Written in `Org-mode`

Musa Al-hassy

2018-07-25

Contents

Arabic ()

Abstract

Herein I document the configurations I utilise with Emacs. Of note are:
This is a literate programming setup.
I have a variety of cosmetics such as showing battery life and flashing upon errors.

- Greating message "Welcome to Emacs 26.1 with Org-mode 9.2.2".

Discoverability capabilities.

- Spacemacs look and feel with completion frameworks `Helm` and `which-key`.

Utilities for working with org-mode files, namely extracting `#+KEYWORD: VALUE` pairs.
As a literate program file with Org-mode, I am ensured optimal navigation through my ever growing configuration files, ease of usability and reference for peers, and, most importantly, better maintainability for myself!
Dear reader when encountering a foregin command `X` I encourage you to execute `(describe-symbol 'X)` ; an elementary Elisp Cheat Sheet can be found here.

1 Why Emacs?

Emacs is a flexible platform for developing end-user applications –unfortunately it is generally perceived as merely a text editor. Some people use it specifically for one or two applications! For example, writers use it as an interface for Org-mode and others use it as an interface for version control with Magit. Org is an organisation tool that can be used for typesetting which subsumes LaTeX, generating many different formats –html, latex, pdf, etc– from a single source, keeping track of schedules, blogging, habit tracking, personal information management tool, and much more. Moreover, its syntax is so natural that most people use it without even knowing! For me, Org allows me to do literate programming: I can program and document at the same time, with no need to seperate the two tasks and with the ability to generate multiple formats and files from a single file. Of course it comes with the basic features of a text editor, but it is much more; for example, it comes with a powerful notion of 'undo': Basic text editors have a single stream of undo, yet in Emacs, we have a tree –when we undo and make new edits, we branch off in our editing stream as if our text was being version controlled as we type! –We can even switch between such branches!

```
;; Allow tree-semantics for undo operations.
(package-install 'undo-tree)
(undo-tree-mode)

;; Execute (undo-tree-visualize) then navigate along the tree to witness
;; changes being made to your file live!
```

Emacs is an extensible editor: You can make it into the editor of your dreams! You can make it suited to your personal needs. If there's a feature you would like, a behaviour your desire, you can simply code that into Emacs with a bit of Lisp. As a programming language enthusiast, for me Emacs is my default Lisp interpreter and a customisable IDE that I use for other programming languages –such as C, Haskell, Agda, Racket, and Prolog. Moreover, being a Lisp interpreter, we can alter the look and feel of Emacs live, without having to restart it –e.g., press `C-x C-e` after the final parenthesis of `(scroll-bar-mode 0)` to run the code that removes the scroll-bar.

> *I use Emacs every day. I rarely notice it. But when I do, it usually brings me joy.* Norman Walsh
>
> If you are a professional writer... Emacs outshines all other editing software in approximately the same way that the noonday sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish. —Neal Stephenson

I have used Emacs as an interface for developing cheat sheets, for making my blog, and as an application for 'interactively learning C'. If anything Emacs is more like an OS than just a text editor –"living within Emacs" provides an abstraction over whatever operating system my machine has: It's so easy to take everything with me. Moreover, the desire to mould Emacs to my needs has made me a better programmer: I am now a more literate programmer and, due to Elisp's documentation-oriented nature, I actually take the time and effort to make meaningful documentation –even though the project it private and will likely only be seen by me.

> *Seeing Emacs as an editor is like seeing a car as a seating-accommodation.* – Karl Voit

Consider reading

- A CEO's Guide to Emacs –A non-programmer introduction to Emacs (••)
- "When did you start using Emacs" discussion on Reddit
- The Emacs Mini Manual, or
- "How to Learn Emacs"
- The Org-mode Reference Manual or Worg: Community-Written Docs which includes a meta-tutorial.

2 TODO Booting Up

2.1 `~/.emacs` vs. `init.org`

Why not keep Emac's configurations in the `~/.emacs` file? This is because the Emacs system may explicitly add, or alter, code in it.

For example, execute the following

1. `M-x customize-variable RET line-number-mode RET`
2. Then press: `toggle`, `state`, then `1`.
3. Now take a look: `(find-file "~/.emacs")`

Notice how additions to the file have been created by 'custom'.

As such, I've chosen to write my Emacs' initialisation configurations in a file named `~/.emacs.d/init.org`: I have a literate configuration which is then loaded using org-mode's tangling feature. Read more about Emacs' initialisation configurations here.

Off topic, I love tiling window managers and had been using xmonad until recently when I obtained a mac machine and now use Amethyst – "Tiling window manager for macOS along the lines of xmonad."

2.2 Elementary Version Control

Soft links are pointers to other filenames, whereas hardlinks are pointers to memory location of a given filename! Soft links are preferable since they defer to the orginal filename and can work across servers.

We can declare them as follows,

```
ln -s source_file myfile
```

If `repo` refers to a directory under version control –or Dropbox– we move our init file and emacs directory to it, then make soft links to these locations so that whenever `~/.emacs` is accessed it will refer to `repo/.emacs` and likewise for `.emacs.d` :-)

On a new machine, copy-paste any existing emacs configs we want to the `repo` folder then `rm -rf ~~/.emacs*` and then make the soft links only.

```
repo=~/Dropbox     ## or my git repository: ~/dotfiles

cd ~

mv .emacs $repo/
ln -s $repo/.emacs .emacs

mv .emacs.elc $repo/
ln -s $repo/.emacs.elc .emacs.elc

mv .emacs.d/ $repo/
ln -s $repo/.emacs.d/ .emacs.d
```

Note the extra `/` after `.emacs.d`!

You may need to unlink soft links if you already have them; e.g., `unlink .emacs.d`.

To make another softlink to a file, say in a blogging directory, we `cd` to the location of interest then execute, say: `ln -s $repo/.emacs.d/init.org init.org`

While we're at it, let's make this super-duper file (and another) easily accessible –since we'll be altering it often–:

```
cd ~

ln -s dotfiles/.emacs.d/init.org init.org
ln -s alhassy.github.io/content/AlBasmala.org AlBasmala.org
```

Below I'll equip us with an Emacs 'porcelain' interface to git –it makes working with version control tremendously convenient. Moreover, I add a little pop-up so that I don't forget to commit often!

2.3 What's in, or at the top of, my `~/.emacs`

We evaluate every piece of emacs-lisp code available here when Emacs starts up by placing the following at the top of our `.emacs` file:

```
;; March 7, 2019 For some reason, I need these here or my org-mode defaults to an older version.
(require 'package)
(add-to-list 'package-archives '("org" . "http://orgmode.org/elpa/"))
(package-initialize)
(require 'org-tempo)

(org-babel-load-file "~/.emacs.d/init.org")
;;
;; My Emacs settings: (find-file "~/.emacs.d/init.org")
```

(I do not generate my `.emacs` file from this source code in-fear of overriding functionality inserted by `custom`.)

Our `.emacs` should be byte-compiled so that when we start Emacs it will automatically determine if the `init.org` file has changed and if so it would tangle it producing the `init.el` file which will then be loaded immediately.

2.4 `use-package` –The start of `init.el`

There are a few ways to install packages –run `C-h C-e` for a short overview. The easiest, for a beginner, is to use the command `package-list-packages` then find the desired package, press `i` to mark it for installation, then install all marked packages by pressing `x`.

Alternatively, one uses the declarative configuration tool use-package –a meta-package that manages other packages and the way they interact.

Background: Recently I switched to mac –first time trying the OS. I had to do a few `package-install`'s and it was annoying. I'm looking for the best way to package my Emacs installation –inlcuding my installed pacakages and configuration– so that I can quickly install it anywhere, say if I go to another machine. It seems `use-package` allows me to configure and auto install packages. On a new machine, when I clone my `.emacs.d` and start emacs, on the first start it should automatically install and compile all of my packages through `use-package` when it detects they're missing.

First we need the basic `package` module which not only allows us to obtain `use-package` but acts as its kernel.

```
;; Make all commands of the "package" module present.
(require 'package)

;; Speef up start up by not loading any packages at startup.
;; (setq package-enable-at-startup nil)
;; Look at the *Messages* buffer before setting this to nil, then after.

;; Internet repositories for new packages.
(setq package-archives '(("org"       . "https://orgmode.org/elpa/")
                         ("gnu"       . "https://elpa.gnu.org/packages/")
                         ("melpa"     . "https://melpa.org/packages/")
                         ("melpa-stable" . "https://stable.melpa.org/packages/")
                         ;; Maintainer is AWOL.
                         ;; ("marmalade" . "https://marmalade-repo.org/packages/")
                         ))

;; Actually get "package" to work.
(package-initialize)
```

We can now:

- `M-x list-packages` to see all melpa packages that can install
  - Not in alphabetical order, so maybe search with `C-s`.
- For example to download the haskell mode: `M-x package-install RET haskell-mode RET`.
  - Or maybe to install `unicode-fonts` ;-)
- Read more at `http://ergoemacs.org/emacs/emacs_package_system.html` or at `https://github.com/milkypostma` `melpa`

We now bootstrap `use-package`,

```
;; Unless it's already installed, update the packages archives,
;; then install the most recent version of "use-package".
(unless (package-installed-p 'use-package)
  (package-refresh-contents)
  (package-install 'use-package))

(require 'use-package)
```

We can now invoke `(use-package XYZ :ensure t)` which should check for the `XYZ` package and make sure it is accessible. If not, the `:ensure t` part tells `use-package` to download it –using `package.el`– and place it somewhere accessible, in `~/.emacs.d/elpa/` by default.

Here's an example use of `use-package`. Below I have my "show recent files pop-up" command set to `C-x C-r`; but what if I forget? This mode shows me all key completions when I type `C-x`, for example. Moreover, I will be shown other commands I did not know about! Neato :-)

```
;; Making it easier to discover Emacs key presses.
(use-package which-key
 :ensure t
 :diminish which-key-mode
 :init (which-key-mode)
 :config (which-key-setup-side-window-bottom)
         (setq which-key-idle-delay 0.05)
)
```

Here are other packages that I want to be installed onto my machine.

```
;; (package-refresh-contents)     ;; Always stay up to date.

;; Nice looking theme ^_^

;; this gives me an error for some reason
;; (use-package spacemacs-theme :ensure t)

;; "C-x" t to toggle between light and dark themes.
(setq my/theme 'spacemacs-light)
(load-theme my/theme t)
(defun my/toggle-theme () "Toggle between dark and light themes."
  (interactive)
  (setq my/theme (if (equal my/theme 'spacemacs-light) 'spacemacs-dark 'spacemacs-light))
  (load-theme my/theme t)
)
(global-set-key "\C-x\ t" 'my/toggle-theme)
```

The Doom Themes also look rather appealing. A showcase of many themes can be found here.

```
;; Efficient version control.
(use-package magit
  :ensure t
  :config (global-set-key (kbd "C-x g") 'magit-status)
)

(use-package htmlize :ensure)
;; Main use: Org produced htmls are coloured.
;; Can be used to export a file into a coloured html.

(use-package biblio  :ensure)     ;; Quick BibTeX references, sometimes.

;; Get org-headers to look pretty! E.g., * → , ** ○, ***
;; https://github.com/emacsorphanage/org-bullets
(use-package org-bullets :ensure t)
(add-hook 'org-mode-hook 'org-bullets-mode)

(use-package haskell-mode :ensure)

;; (use-package flycheck          ;; Dynamic syntax checking.
;;  :init (global-flycheck-mode) :ensure)

(use-package dash :ensure)    ;; "A modern list library for Emacs"
(use-package s    :ensure)    ;; "The long lost Emacs string manipulation library".
```

Note:

- dash: "A modern list library for Emacs"
  - E.g., `(--filter (> it 10) (list 8 9 10 11 12))`
- s: "The long lost Emacs string manipulation library".
  - E.g., `s-trim, s-replace, s-join`.

Finally, since I've symlinked my `.emacs`:

```
;; Don't ask for confirmation when opening symlinked files.
(setq vc-follow-symlinks t)
```

2.5 `magit` –Emacs' porcelain interface to git

Why use `magit` as the interface to the git version control system? In a magit buffer nearly everything can be acted upon: Press `return`, or `space`, to see details and `tab` to see children items, usually.

Below is my personal quick guide to working with magit. A quick magit tutorial can be found on jr0cket's blog

`magit-init` Put a project under version control. The mini-buffer will prompt you for the top level folder version. A `.git` folder will be created there.

`magit-status` , `C-x g` See status in another buffer. Press `?` to see options, including:

`q` Quit magit, or go to previous magit screen.

`s` Stage, i.e., add, a file to version control. Add all untracked files by selecting the *Untracked files* title.

`k` Kill, i.e., delete a file locally.

`K` This' `(magit-file-untrack)` which does `git rm --cached`.

`i` Add a file to the project `.gitignore` file. Nice stuff =)

`u` Unstage a specfif staged change highlighed by cursor. `C-u s` stages everything –tracked or not.

`c` Commit a change.

- A new buffer for the commit message appears, you write it then commit with `C-c C-c` or otherwise cancel with `C-c C-k`. These commands are mentioned to you in the minibuffer when you go to commit.
- You can provide a commit to *each* altered chunk of text! This is super neat, you make a series of local such commits rather than one nebulous global commit for the file. The `magit` interface makes this far more accessible than a standard terminal approach!
- You can look at the unstaged changes, select a *region*, using `C-SPC` as usual, and commit only that if you want!
- When looking over a commit, `M-p/n` to efficiently go to previous or next altered sections.
- Amend a commit by pressing `a` on `HEAD`.

`d` Show differences, another `d` or another option.

- This is magit! Each hunk can be acted upon; e.g., `s` or `c` or `k` ;-)
- The staging area is akin to a pet store; commiting is taking the pet home.

`v` Revert a commit.

`x` Undo last commit. Tantamount to `git reset HEAD~` when cursor is on most recent commit; otherwise resets to whatever commit is under the cursor.

`l` Show the log, another `l` for current branch; other options will be displayed.

- Here `space` shows details in another buffer while cursour remains in current buffer and, moreover, continuing to press `space` scrolls through the other buffer! Neato.

`P` Push.

`F` Pull.

`:` Execute a raw git command; e.g., enter `whatchanged`.

The status buffer may be refereshed using `g`, and all magit buffer by `G`.

Press `tab` to see collapsed items, such as what text has been changed.

Notice that every time you press one of these commands, a 'pop-up' of realted git options appears! Thus not only is there no need to memorize many of them, but this approach makes discovering other commands easier.

Use `M-x (magit-list-repositories) RET` to list local repositories:

Below are the git repos I'd like to clone

```
;; Do not ask about this variable when cloning.
(setq magit-clone-set-remote.pushDefault t)

(defun maybe-clone (remote local)
  "Clone a 'remote' repository if the 'local' directory does not exist.
    Yields 'nil' when no cloning transpires, otherwise yields ''cloned-repo''.
  "
  (unless (file-directory-p local)
     (magit-clone remote local)
     (add-to-list 'magit-repository-directories '(,local    . 0))
     'cloned-repo)
)

;; Set variable without asking.
(setq magit-clone-set-remote.pushDefault 't)

;; Public repos
(maybe-clone "https://github.com/alhassy/dotfiles" "~/dotfiles")
(maybe-clone "https://github.com/alhassy/alhassy.github.io" "~/alhassy.github.io")
(maybe-clone "https://github.com/alhassy/CheatSheet" "~/CheatSheet")
(maybe-clone "https://github.com/alhassy/ElispCheatSheet" "~/ElispCheatSheet")
(maybe-clone "https://github.com/alhassy/MyUnicodeSymbols" "~/MyUnicodeSymbols")
(maybe-clone "https://github.com/alhassy/interactive-way-to-c" "~/interactive-way-to-c")

;; Private repos
;; (maybe-clone "https://gitlab.cas.mcmaster.ca/carette/cs3fp3.git" "~/3fp3") ;; cat adventures
;; (maybe-clone "https://gitlab.cas.mcmaster.ca/RATH/RATH-Agda"     "~/RATH-Agda")
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/assignment-distribution.git" "~/3ea3/assig
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/notes.git" "~/3ea3/notes")
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/assignment-development.git" "~/3ea3/assign
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/kandeeps.git" "~/3ea3/sujan")
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/horsmane.git" "~/3ea3/emily")
(maybe-clone "https://gitlab.cas.mcmaster.ca/3ea3-winter2019/anderj12.git" "~/3ea3/jacob")
;; (maybe-clone "https://gitlab.cas.mcmaster.ca/alhassm/3EA3.git" "~/3ea3/_2018")
;; (maybe-clone "https://gitlab.cas.mcmaster.ca/2DM3/LectureNotes.git" "~/2dm3")

;; Likely want to put a hook when closing emacs, or at some given time,
;; to show me this buffer so that I can 'push' if I haven't already!
;
; (magit-list-repositories)
```

Let's always notify ourselves of a file that has uncommited changes –we might have had to step away from the computer and forgotten to commit.

```
(require 'magit-git)

(defun my/magit-check-file-and-popup ()
  "If the file is version controlled with git
  and has uncommitted changes, open the magit status popup."
  (let ((file (buffer-file-name)))
    (when (and file (magit-anything-modified-p t file))
      (message "This file has uncommited changes!")
      (when nil ;; Became annyoying after some time.
      (split-window-below)
      (other-window 1)
      (magit-status)))))

;; I usually have local variables, so I want the message to show
;; after the locals have been loaded.
(add-hook 'find-file-hook
  '(lambda ()
      (add-hook 'hack-local-variables-hook 'my/magit-check-file-and-popup)
   ))
```

Let's try this out:

```
(progn (eshell-command "echo change-here >> ~/dotfiles/.emacs")
       (find-file "~/dotfiles/.emacs")
)
```

In doubt, execute `C-h e` to jump to the `*Messages*` buffer.

2.6 Fix spelling as you type –and a thesaurus too!

I would like to check spelling by default.

`C-;` Cycle through corrections for word at point.

`M-$` Check and correct spelling of the word at point

`M-x ispell-change-dictionary RET TAB` To see what dictionaries are available.

Flyspell needs a spell checking tool, which is not included in Emacs. We install `aspell` spell checker using, say, homebrew via `brew install aspell`. Note that Emacs' `ispell` is the interface to such a command line spelling utility.

```
(setq ispell-program-name "/usr/local/bin/aspell")


(setq ispell-dictionary "en_GB") ;; set the default dictionary

(add-hook 'text-mode-hook 'flyspell-mode)
```

Ecnabling fly-spell for text-mode enables it for org and latex modes since they derive from text-mode.
Let us select a correct spelling merely by clicking on a word.

```
(eval-after-load "flyspell"
  ' (progn
     (define-key flyspell-mouse-map [down-mouse-3] #'flyspell-correct-word)
     (define-key flyspell-mouse-map [mouse-3] #'undefined)))
```

Colour incorrect works; default is an underline.

```
(global-font-lock-mode t)
(custom-set-faces '(flyspell-incorrect ((t (:inverse-video t)))))
```

Finally, save to user dictionary without asking:

```
(setq ispell-silently-savep t)
```

Nowadays, I very rarely write non-literate programs, but if I do I'd like to check spelling only in comments/strings. E.g.,

```
(add-hook           'c-mode-hook 'flyspell-prog-mode)
(add-hook 'emacs-lisp-mode-hook 'flyspell-prog-mode)
```

Use the thesaurus Emacs frontend Synosaurus to avoid unwarranted repetition.

```
(use-package synosaurus
  :ensure t
  :diminish synosaurus-mode
  :init    (synosaurus-mode)
  :config  (setq synosaurus-choose-method 'popup) ;; 'ido is default.
           (global-set-key (kbd "M-#") 'synosaurus-choose-and-replace)
)
```

The thesaurus is powered by the Wordnet `wn` tool, which can be invoked without an internet connection!

```
;; (shell-command "brew cask install xquartz &") ;; Dependency
;; (shell-command "brew install wordnet &")
```

Use this game to help you learn to spell words that you're having trouble with; see `~/Dropbox/spelling.txt`.

```
(autoload 'typing-of-emacs "~/.emacs.d/typing.el" "The Typing Of Emacs, a game." t)
```

Practice touch typing using speed-type.

```
(use-package speed-type :ensure t)
```

Running `M-x speed-type-region` on a region of text, or `M-x speed-type-buffer` on a whole buffer, or just `M-x speed-type-text` will produce the selected region, buffer, or random text for practice. The timer begins when the first key is pressed and stats are shown when the last letter is entered.

2.7 Unicode Input via Agda Input

Agda is one of my favourite languages, it's like Haskell on steroids. Let's set it up.

Executing `agda-mode setup` appends the following text to the `.emacs` file. Let's put it here ourselves.

```
(load-file (let ((coding-system-for-read 'utf-8))
                (shell-command-to-string "/usr/local/bin/agda-mode locate")))
```

I almost always want the `agda-mode` input method.

```
(require 'agda-input)
(add-hook 'text-mode-hook (lambda () (set-input-method "Agda")))
(add-hook 'org-mode-hook (lambda () (set-input-method "Agda")))
```

Below are my personal Agda input symbol translations; e.g., `\set` → ∋. Note that we could give a symbol new Agda TEX binding interactively: `M-x customize-variable agda-input-user-translations` then `INS` then for key sequence type `set` then `INS` and for string paste ∋.

```
;; category theory
(add-to-list 'agda-input-user-translations '("set" ""))
(add-to-list 'agda-input-user-translations '("alg" ""))
(add-to-list 'agda-input-user-translations '("split" ""))
(add-to-list 'agda-input-user-translations '("join" ""))
(add-to-list 'agda-input-user-translations '("adj" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))
(add-to-list 'agda-input-user-translations '(";;" ""))

;; lattices
(add-to-list 'agda-input-user-translations '("meet" ""))
(add-to-list 'agda-input-user-translations '("join" ""))

;; residuals
(add-to-list 'agda-input-user-translations '("syq"  ""))
(add-to-list 'agda-input-user-translations '("over" ""))
(add-to-list 'agda-input-user-translations '("under" ""))
        ;; Maybe "\\" shortcut?

;; Z-quantification range notation, e.g., " x  R • P"
(add-to-list 'agda-input-user-translations '("|" ""))

;; adjunction isomorphism pair
(add-to-list 'agda-input-user-translations '("floor"  ""))
(add-to-list 'agda-input-user-translations '("lower"  ""))
(add-to-list 'agda-input-user-translations '("lad"    ""))
(add-to-list 'agda-input-user-translations '("ceil"   ""))
(add-to-list 'agda-input-user-translations '("raise"  ""))
(add-to-list 'agda-input-user-translations '("rad"    ""))

;; silly stuff
;;
;; angry, cry, why-you-no
(add-to-list 'agda-input-user-translations
   '("whyme" "()" "_" "()"))
;; confused, disapprove, dead, shrug
(add-to-list 'agda-input-user-translations
   '("what" "(°°)" "(_)" "()" "¯\\_()_/¯"))
;; dance, csi
(add-to-list 'agda-input-user-translations
   '("cool" "(-_-)(-_-)(-_- )" "•_•)
( •_•)>-
(_)
"))
;; love, pleased, success, yesss
(add-to-list 'agda-input-user-translations
   '("smile" "" "()" "(••)" "(_)"))
```

Finally let's effect such translations.

```
;; activate translations
(agda-input-setup)
```

Note that the effect of Emacs unicode input could be approximated using `abbrev-mode`.

2.8 Locally `toggle` a variable

It is dangerous to load a file with local variables; instead we should load files without evaluating locals, read the locals to ensure they are safe –e.g., there's nothing malicious like `eval: (delete-file your-important-file.txt)`– then revert the buffer to load the locals.

However, when preprocessing my own files I sometimes wish to accept all locals without being queried and so have the following combinator.

```
(defmacro toggle (variable value code)
  "Locally set the value of ‘variable’ to be ‘value’ in the scope of ‘code’.
   In particular, the value of ‘variable’, if any, *is* affected
   to produce useful sideffects. It retains its orginal value outside this call.

   Example uses include terse replacements for one-off let-statements,
   or, more likely, of temporarily toggeling important values, such as
   ‘kill-buffer-query-functions’ for killing a process buffer without confirmation.

   Another example: ‘(toggle enable-local-variables :all )’ to preprocess files
   without being queried about possibly dangerous local variables.
  "
  `(let ((_initial_value_ ,variable))
        (setq ,variable ,value)
        ,code
        (setq ,variable _initial_value_)
  )
)
```

Since emacs-lisp interprets definitions sequentially, I define `toggle` here since I employ it in the next section.

2.9 TODO Altering PATH

```
;; https://emacs.stackexchange.com/questions/4090/org-mode-cannot-find-pdflatex-using-mac-os

(defun set-exec-path-from-shell-PATH ()
  "Sets the exec-path to the same value used by the user shell"
  (let ((path-from-shell
         (replace-regexp-in-string
          "[[:space:]\n]*$" ""
          (shell-command-to-string "$SHELL -l -c 'echo $PATH'"))))
    (setenv "PATH" path-from-shell)
    (setq exec-path (split-string path-from-shell path-separator))))

;; call function now
(set-exec-path-from-shell-PATH)
```

3 Cosmetics

3.1 Startup message: Emacs & Org versions

```
;; Silence the usual message: Get more info using the about page via C-h C-a.
(setq inhibit-startup-message t)

(defun display-startup-echo-area-message ()
  (message
```

```
      (concat "Welcome! Emacs "  emacs-version
              "; Org-mode "      org-version
              "; on system "     (system-name)
      )
  )
)

;; (setq initial-scratch-message "Welcome! This' the scratch buffer" )
```

3.2 Flashing when something goes wrong

Make top and bottom of screen flash when something unexpected happens thereby observing a warning message in the minibuffer. E.g., C-g, or calling an unbound key sequence, or misspelling a word.

```
(setq visible-bell 1)
;; Enable flashing mode-line on errors
;; On MacOS, this shows a caution symbol ^_^
```

3.3 My to-do list: The initial buffer when Emacs opens up

```
(find-file "~/Dropbox/todo.org")
;; (setq initial-buffer-choice "~/Dropbox/todo.org")

(split-window-right)                          ;; C-x 3
(other-window 1)                                  ;; C-x 0
(toggle enable-local-variables 'all           ;; Load *all* locals.
  (find-file "~/.emacs.d/init.org"))
```

(describe-symbol 'enable-local-variables)

3.4 Showing date, time, and battery life

```
(setq display-time-day-and-date t)
(display-time)
(display-battery-mode 1)
```

3.5 Hiding Scrollbar, tool bar, and menu

```
(tool-bar-mode -1)
(scroll-bar-mode -1)
(menu-bar-mode -1)
```

3.6 Increase/decrease text size and word wrapping

```
(global-set-key (kbd "C-+") 'text-scale-increase)
(global-set-key (kbd "C--") 'text-scale-decrease)
  ;; C-x C-0 restores the default font size

;; Truncate lines in all buffers
(setq-default truncate-lines t)
(setq-default global-visual-line-mode t)
```

3.7 Delete Selection mode

Delete Selection mode lets you treat an Emacs region much like a typical text selection outside of Emacs: You can replace the active region. We can delete selected text just by hitting the backspace key.

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

3.8 Highlight & complete parenthesis pair when cursor is near ;-)

```
(use-package smartparens
  :ensure t
  :init
  (smartparens-global-mode 1)
  (show-smartparens-global-mode +1)

  :bind (;; ("M-n" . sp-next-sexp)
         ;; ("M-p" . sp-previous-sexp)
         ("M-f" . sp-forward-sexp)
         ("M-b" . sp-backward-sexp)
         )

  :config
  ;; Enable smartparens everywhere
  (use-package smartparens-config)

  (setq
   ;; smartparens-strict-mode t
   ;; sp-autoinsert-if-followed-by-word t
   ;; sp-autoskip-closing-pair 'always
   ;;sp-base-key-bindings 'paredit
   sp-hybrid-kill-entire-symbol nil)

  ;; In Elisp mode, do not `close' a back-tick!
  (sp-local-pair 'emacs-lisp-mode "`" nil :when '(sp-in-string-p))
  )
```

3.9 Minibuffer should display line and column numbers

```
(global-display-line-numbers-mode t)
; (line-number-mode t)
(column-number-mode t)
```

3.10 Completion Frameworks

Helm provides possible completions and also shows recently executed commands when pressing `M-x`.

Extremely helpful for when switching between buffers, `C-x b`, and discovering & learning about other commands! E.g., press `M-x` to see recently executed commands and other possible commands!

Try and be grateful.

```
(use-package helm
 :ensure t
 :init (helm-mode t)
 :bind
  ("C-x C-r" . helm-recentf)      ; search for recently edited
```

```
  ;; Helm provides generic functions for completions to replace
  ;; tab-completion in Emacs with no loss of functionality.
  ("M-x" . 'helm-M-x)
  ("C-x r b" .'helm-filtered-bookmarks)
  ("C-x C-f" . 'helm-find-files)
)
;; (global-set-key (kbd "M-x") 'execute-extended-command) ;; Default "M-x"

;; Yet, let's keep tab-completetion anyhow.
(define-key helm-map (kbd "TAB") #'helm-execute-persistent-action)
(define-key helm-map (kbd "<tab>") #'helm-execute-persistent-action)
;; We can list 'actions' on the currently selected item by C-z.
(define-key helm-map (kbd "C-z")  'helm-select-action)
```

When `helm-mode` is enabled, even help commands make use of it. E.g., `C-h o` runs `describe-symbol` for the symbol at point, and `C-h w` runs `where-is` to find the key binding of the symbol at point. Both show a pop-up of other possible commands.

Incidentally, helm even provides an interface for the top program via `helm-top`. It also serves as an interface to popular search engines and over 100 websites such as `google, stackoverflow`, and `arxive`.

```
;; (shell-command "brew install surfraw &")
;;
;; Invoke helm-surfraw
```

If we want to perform a google search, with interactive suggestions, then invoke `helm-google-suggest` –which can be acted for other serves, such as Wikipedia or Youtube by `C-z`. For more google specific options, there is the `google-this` package.

Let's switch to a powerful searching mechanism – helm-swoop. It allows us to not only search the current buffer but also the other buffers and to make live edits by pressing `C-c C-e` when a search buffer exists. Incidentally, executing `C-s` on a word, region, will search for that particular word, region; then apply changes by `C-x C-s`.

```
(use-package helm-swoop
  :ensure t
  :bind
  (
   ("C-s"     . 'helm-swoop)           ;; search current buffer
   ("C-M-s"   . 'helm-multi-swoop-all) ;; Search all buffer
   ;; Go back to last position where 'helm-swoop' was called
   ("C-S-s" . 'helm-swoop-back-to-last-point)
  )
 :config
   ;; Give up colour for speed.
  (setq helm-swoop-speed-or-color nil)
)
```

Press `M-i` after a search has executed to enable it for all buffers.

We can also limit our search to org files, or buffers of the same mode, or buffers belonging to the same project!

Finally, let's enable "complete anything" mode.

```
(use-package company
  :config
  (add-hook 'after-init-hook 'global-company-mode))

(global-set-key (kbd "C-c h") 'company-complete)
```

Note that `Meta-/` goes through a sequence of completions.

3.11 On the fly syntax checking

```
(use-package flycheck
  :defer 2
  :init (global-flycheck-mode)
  :custom
  (flycheck-display-errors-delay .3))
```

3.12 "FIXME:" Keyword

In the **middle** of a senctence, I need a FIXME: Woah!

```
;; "FIXME:" is now a keyword, and so will be highlighted
(font-lock-add-keywords nil
'(("\\<\\(FIXME:\\)" 1
font-lock-warning-face t)))
;;
;; Src: https://www.gnu.org/software/emacs/manual/html_node/emacs/Font-Lock.html#Font-Lock
;; Also: https://stackoverflow.com/a/756856/3550444
```

3.13 Neotree: Directory Tree Listing

We open a nifty file manager upon startup.

```
;; neotree --sidebar for project file navigation
(use-package neotree :ensure t
  :config (global-set-key "\C-x\ d" 'neotree-toggle))

(use-package all-the-icons :ensure t)
;; Only do this once: (all-the-icons-install-fonts)


(setq neo-theme 'icons)
(neotree-refresh)

;; Open it up upon startup.
(neotree-toggle)
```

By default `C-x d` invokes `dired`, but I prefer `neotree` for file management.
Useful navigational commands include

- `U` to go up a directory.
- `C-c C-c` to change directory focus; `C-C c` to type the directory out.
- `?` or `h` to get help and `q` to quit.

As always, to go to the neotree pane when it's the only other window, execute `C-x o`.

3.14 Window resizing using the golden ratio DISABLED

Let's load the following package, which automatically resizes windows so that the window containing the cursor is the largest, according to the golden ratio. Consequently, the window we're working with is nice and large yet the other windows are still readable.

```
(use-package golden-ratio
  :ensure t
  :diminish golden-ratio-mode
  :init (golden-ratio-mode 1))
```

After some time this got a bit annoying and I'm no longer using this.

3.15 Jump between windows using Cmd+Arrow

```
(use-package windmove
  :ensure t
  :config
  ;; use command key on Mac
  (windmove-default-keybindings 'super)
  ;; wrap around at edges
  (setq windmove-wrap-around t))
```

4 Helpful Functions & Shortcuts

Here is a collection of Emacs-lisp functions that I have come to use in other files.

Let's save a few precious seconds,

```
;; change all prompts to y or n
(fset 'yes-or-no-p 'y-or-n-p)
```

4.1 Bind `recompile` to `C-c C-m` – "m" for "m"ake

```
(defvar my-keys-minor-mode-map
  (let ((map (make-sparse-keymap)))
    (define-key map (kbd "C-c C-m") 'recompile)
    map)
  "my-keys-minor-mode keymap.")

(define-minor-mode my-keys-minor-mode
  "A minor mode so that my key settings override annoying major modes."
  :init-value t
  :lighter " my-keys")

(my-keys-minor-mode)
```

4.2 Reload buffer with `f5`

I do this so often it's not even funny.

```
(global-set-key [f5] '(lambda () (interactive) (revert-buffer nil t nil)))
```

In Mac OS, one uses `Cmd-r` to reload a page and spacemacs binds buffer reversion to `Cmd-u` –in Emacs, Mac's `Cmd` is referred to as the 'super key' and denoted `s`.

Moreover, since I use Org-mode to generate code blocks and occasionally inspect them, it would be nice if they automatically reverted when they were regenerated –Emacs should also prompt me if I make any changes!

```
;; Auto update buffers that change on disk.
;; Will be prompted if there are changes that could be lost.
(global-auto-revert-mode 1)
```

4.3 Kill to start of line

Dual to `C-k`,

```
;; M-k kills to the left
(global-set-key "\M-k" '(lambda () (interactive) (kill-line 0)) )
```

4.4 `file-as-list` and `file-as-string`

```
(defun file-as-list (filename)
  "Return the contents of FILENAME as a list of lines"
  (with-temp-buffer
    (insert-file-contents filename)
    (split-string (buffer-string))))

(defun file-as-string (filename)
  "Return the contents of FILENAME as a list of lines"
  (with-temp-buffer
    (insert-file-contents filename)
    (buffer-string)))
```

4.5 `kill-other-buffers`

```
(defun kill-other-buffers ()
  "Kill all other buffers."
  (interactive)
  (mapc 'kill-buffer (delq (current-buffer) (buffer-list))))
```

4.6 `create-scratch-buffer`

```
;; A very simple function to recreate the scratch buffer:
;; ( http://emacswiki.org/emacs/RecreateScratchBuffer )
(defun create-scratch-buffer nil
   "create a scratch buffer"
   (interactive)
   (switch-to-buffer (get-buffer-create "*scratch*"))
   (lisp-interaction-mode))
```

4.7 Switching from 2 horizontal windows to 2 vertical windows

I often find myself switching from a horizontal view of two windows in Emacs to a vertical view. This requires a variation of `C-x 1 RET C - x 3 RET C-x o X-x b RET`. Instead I now only need to type `C-|` to make this switch.

```
(defun ensure-two-vertical-windows ()
  "hello"
 (interactive)
 (other-window 1)                        ;; C-x 0
 (let ((otherBuffer (buffer-name)))
   (delete-window)                        ;; C-x 0
   (split-window-right)                        ;; C-x 3
   (other-window 1)                       ;; C-x 0
   (switch-to-buffer otherBuffer)        ;; C-x b RET
 )
 (other-window 1)
)
(global-set-key (kbd "C-|") 'ensure-two-vertical-windows)
```

4.8 `re-replace-in-file`

```
(defun re-replace-in-file (file regex whatDo) "Find and replace a regular expression in-place in a file

    (find-file file)
```

```
    (goto-char 0)
    (let ((altered (replace-regexp-in-string regex whatDo (buffer-string))))
      (erase-buffer)
      (insert altered)
      (save-buffer)
      (kill-buffer)
   )
)
```

Example usage:

```
;; Within mysite.html we rewrite: <h1.*h1>      <h1.*h1>\n NICE
;; I.e., we add a line break after the first heading and a new word, ‘‘NICE’’.
(re-replace-in-file "mysite.html"
                    "<h1.*h1>"
                    (lambda (x) (concat x "\n NICE")))
```

4.8.1 `mapsto`: Simple rewriting for current buffer

```
(defun mapsto (this that)
  "In the current buffer make the regular expression rewrite: this  that."
  (let* ((current-location (point))
       ;; Do not alter the case of the <replacement text>.
       (altered (replace-regexp-in-string this (lambda (x) that) (buffer-string) 'no-fixed-case))
       )
      (erase-buffer)
      (insert altered)
      (save-buffer)
      (goto-char current-location)
  )
)
```

4.9 Obtaining Values of `#+KEYWORD` Annotations

Org-mode settings are, for the most part, in the form `#+KEYWORD: VALUE`. Of notable interest are the `TITLE` and `NAME` keywords. We use the following `org-keywords` function to obtain the values of arbitrary `#+THIS : THAT` pairs, which may not necessarily be supported by native Org-mode –we do so for the case, for example, of the `CATEGORIES` and `IMAGE` tags associated with an article.

```
;; Src: http://kitchingroup.cheme.cmu.edu/blog/2013/05/05/Getting-keyword-options-in-org-files/
(defun org-keywords ()
  "Parse the buffer and return a cons list of (property . value) from lines like: #+PROPERTY: value"
  (org-element-map (org-element-parse-buffer 'element) 'keyword
                   (lambda (keyword) (cons (org-element-property :key keyword)
                                           (org-element-property :value keyword)))))

(defun org-keyword (KEYWORD)
  "Get the value of a KEYWORD in the form of #+KEYWORD: value"
  (cdr (assoc KEYWORD (org-keywords))))
```

Note that capitalisation in a "#+KeyWord" is irrelevant.
See here on how to see the abstract syntax tree of an org file and how to manipulate it.

4.10 Quickly pop-up a terminal, run a command, close it

```
(defvar *toggle-terminal* t
   "
   The variable to decide whether a terminal has been pushed
   and now needs to be popped, or vice-versa.
   By default, open a terminal.
   " )

(defun toggle-terminal ()
   "Pop up a terminal, do some work, then close it using the same command.
   "
   (interactive)
   (if *toggle-terminal*

        (progn
               (split-window-right)
               (other-window 1)
               (eshell))

        (toggle kill-buffer-query-functions nil (kill-buffer eshell-buffer-name))
        (delete-window)
    )
   (setq *toggle-terminal* (not *toggle-terminal*))
)

(global-set-key "\C-t" 'toggle-terminal)
```

4.11 `C-x k` kills current buffer

By default `C-x k` prompts to select which buffer should be selected. I almost always want to kill the current buffer, so let's not waste time making such a tedious decision.

```
;; Kill current buffer; prompt only if
;; there are unsaved changes.
(global-set-key (kbd "C-x k")
  '(lambda () (interactive) (kill-buffer (current-buffer))))
```

5 Org-mode related things

Here is useful Org-Mode Table Editing Cheatsheet.

```
(setq org-ellipsis " ")
```

5.1 Capture

Capture lets me quickly make notes & capture ideas, with associated reference material, without any interruption to the current work flow.

E.g., I have a task, or something I wish to note down, rather than opening some file, then making a heading, then writing it; instead, I press `C-c c t` and a pop-up appears, I make my note, and it disappears with my notes file(s) now being altered! Moreover, by default it provide a timestamp and a link to the file location where I made the note –helpful for tasks, tickets, to be tackled later on.

```
(setq org-default-notes-file "~/Dropbox/todo.org")
(define-key global-map "\C-cc" 'org-capture)
```

By default we only get a 'tasks' form of capture, let's add some more.

```
(setq my/capture-template-node "* %?\n:PROPERTIES:\n:CREATED: %U\n:END:\n\n")

(setq org-capture-templates
  `(
     ("f" "Todos needing less than 5 minutes"
       entry (file+headline org-default-notes-file "Five-Minutes Tasks")
       ,my/capture-template-node :empty-lines 1)

     ("t" "Important tasks that need to be tackled; e.g., research."
       entry (file+headline org-default-notes-file "TODO Important Tasks")
       ,my/capture-template-node :empty-lines 1)

     ("e" "Notes related to Emacs (●●)"
       entry (file+headline org-default-notes-file "Emacs")
       ,my/capture-template-node :empty-lines 1)

     ("b" "Notes related to my personal blog"
       entry (file+headline org-default-notes-file "Blog")
       ,my/capture-template-node :empty-lines 1)

     ;; Such a clutter is a terrible idea, hopefully I will have more tags in time.
     ("r" "Arbitrary ideas and notes ()"
       entry (file+headline org-default-notes-file "Random Org-Capture Notes")
       ,my/capture-template-node :empty-lines 1)
))
```

In my `todo.org` file I have `Five-Minutes` as an Org keyword.

For now I capture everything into a single file. One would ideally keep separate client, project, information in its own org file.

5.2 Org Speed Keys

Let's enable the Org Speed Keys so that when the cursor is at the beginning of a headline, we can perform fast manipulation & navigation using the standard Emacs movement controls, such as

- `#` toggle `COMMENT`-ing for an org-header.
- `I/O` clock In/Out to the task defined by the current heading.
  - Keep track of your work times!
  - `v` view agenda.
- `u` for jumping upwards to the parent heading.
- `c` for cycling structure below current heading, or `C` for cycling global structure.
- `i` insert a new same-level heading below current heading.
- `w` refile current heading; options list pops-up to select which heading to move it to. Neato!
- `t` cycle through the available TODO states.
- `^` sort children of current subtree; brings up a list of sorting options.
- `n/p` for next/previous *visible* heading.

- `f/b` for jumping forward/backward to the next/previous *same-level* heading.
- `D/U` move a heading down/up.
- `L/R` recursively promote (move leftwards) or demote (more rightwards) a heading.

Finally, `?` to see a complete list of keys available.

```
(setq org-use-speed-commands t)

;; Add more speed commands by adding to this association list.
;; (describe-symbol 'org-speed-commands-user)
```

5.3 Template expansion (<s Tab, etc.)

In org-mode we type `<X TAB` to obtain environment templates, such as `<s` for source blocks or `<q` for quote blocks. It seems recent changes to the org-mode structure template expansion necessitate explicitly loading `org-tempo`.

```
(require 'org-tempo)
```

To insert source blocks with the assistance of a pop-up: `C-c C-v d` ;-) Perhaps more usefully, invokcaing within a source block splits it up into two seperate blocks! Moreover, if invoked on a selected region, it puts the region into a new code block! Wow!

- `<X` allows you to obtain the org-block assigned to shortcut `x`.
- `C-C C-v C-d` and `C-c C-v d` refer to the `org-babel-demarcate-block`, which provides *source* blocks.
- `C-c C-,` refers to `org-insert-structure-template`, which provides non-source blocks, such as quote `<q` and comment `C`.

5.3.1 <el Emacs-lisp source blocks

`<el` to begin an emacs-lisp source block – `<e` is for an example block.

```
(add-to-list 'org-structure-template-alist
     '("el" . "src emacs-lisp"))

(require 'org-tempo)
;; (defun org--check-org-structure-template-alist (x) "n" t)
```

5.3.2 <ag (Org) Agda source template

```
(add-to-list 'org-structure-template-alist
     '("ag" . "src org-agda"))
```

5.3.3 <hs Haskell source template

```
(add-to-list 'org-structure-template-alist
     '("hs" . "src haskell"))
```

5.3.4 <ic Interactive Way to C source template

```
 (add-to-list 'org-structure-template-alist
      '("ic" . "src c :tangle (currently-working-with \"name\")"))
```

5.3.5 <ich Interactive Way to C header template

```
 (add-to-list 'org-structure-template-alist
      '("ich" . "src c :tangle (currently-working-with-header \"name\")"))
```

5.3.6 `<ver` Verbatim template

```
(add-to-list 'org-structure-template-alist
     '("ver" . "verbatim"))
```

5.4 `ox-extra`: Using `:ignore:` to ignore headings but use the bodies

Use the `:ignore:` tag on headlines you'd like to have ignored, while not ignoring their content –see here.

```
(load "~/dotfiles/.emacs.d/ox-extra.el")
(ox-extras-activate '(ignore-headlines))
```

5.5 Executing code from `src` blocks

For example, to execute a shell command in emacs, write a `src` with a shell command, then `C-c c-c` to see the results. Emacs will generally query you to ensure you're sure about executing the (possibly dangerous) code block; let's stop that:

```
; Seamless use of babel: No confirmation upon execution.
(setq org-confirm-babel-evaluate nil)
```

A worked out example can be obtained as follows: `<g TAB` then `C-c C-C` to make a nice simple graph –the code for this is in the next section.

Some initial languages we want org-babel to support:

```
(org-babel-do-load-languages
  'org-babel-load-languages
  '(
    (emacs-lisp . t)
    ;; (shell          . t)
    (python . t)
    (haskell . t)
    (ruby          . t)
    (ocaml          . t)
    (dot          . t)
    (latex          . t)
    (org          . t)
    (makefile          . t)
    ))

(setq org-src-preserve-indentation t)
```

More languages can be added using `add-to-list`.

5.6 Hiding Emphasise Markers & Inlining Images

```
;; org-mode math is now highlighted ;-)
(setq org-highlight-latex-and-related '(latex))

;; Hide the *,=,/ markers
(setq org-hide-emphasis-markers t)

;; (setq org-pretty-entities t)
;; to have \alpha, \to and others display as utf8 http://orgmode.org/manual/Special-symbols.html

;; Let's set inline images.
```

```
(setq org-display-inline-images t)
(setq org-redisplay-inline-images t)
(setq org-startup-with-inline-images "inlineimages")
```

5.7 Workflow States

Here are some of my common workflow states, –the '!' indicates a timestamp should be generated–

```
(setq org-todo-keywords
      (quote ((sequence "Todo(t)" "Started(s)" "|" "Done(d/!)")
              (sequence "Waiting(w@/!)" "on_hold(h@/!)" "|" "Cancelled(c@/!)")
              (sequence "|" "Five-Minutes(f)") ;; For tasks that take 5 minutes to complete.
             )
      )
)
```

The `@` brings up a pop-up to make a local note about why the state changed. **Super cool stuff!**
Here's how they are coloured,

```
(setq org-todo-keyword-faces
      (quote (("Todo" :foreground "red" :weight bold)
              ("Started" :foreground "blue" :weight bold)
              ("Done" :foreground "forest green" :weight bold)
              ("Waiting" :foreground "orange" :weight bold)
              ("On_hold" :foreground "magenta" :weight bold)
              ("Cancelled" :foreground "forest green" :weight bold))))
```

Now we press `C-c C-t` then the letter shortcut to actually make the state of an org heading.

```
(setq org-use-fast-todo-selection t)
```

We can also change through states using Shift- left, or right.
Let's draw a state diagram to show what such a workflow looks like.
PlantUML supports drawing diagrams in a tremendously simple format –it even supports Graphviz/DOT directly and many other formats. Super simple setup instructions can be found here; below are a bit more involved instructions. Read the manual here.

```
;; Install the tool
; (async-shell-command "brew cask install java") ;; Dependency
; (async-shell-command "brew install plantuml")

;; Tell emacs where it is.
;; E.g., (async-shell-command "find / -name plantuml.jar")
(setq org-plantuml-jar-path
      (expand-file-name "/usr/local/./Cellar/plantuml/1.2019.3/libexec/plantuml.jar"))

;; Enable C-c C-c to generate diagrams from plantuml src blocks.
(add-to-list 'org-babel-load-languages '(plantuml . t) )
;; (require 'ob-plantuml)
```

Let's use this!
Of note:

- Multiline comments are with `/' comment here '/`, single quote starts a one-line comment.
- Nodes don't need to be declared, whose names may contain spaces if they are enclosed in double-quotes.

- One forms an arrow between two nodes by writing a line with `x ->[label here] y` or `y <- x`; or using `-->` and `<--` for dashed lines. The label is optional.

  To enforce a particular layout, use `-X->` where `X  {up, down, right, left}`.

- To declare that a node `x` has fields `d, f` we make two new lines having `x :  f` and `x :  d`.

- One adds a note by a node `x` as follows: `note right of x:  words then newline\nthen more words`. Likewise for notes on the `left, top, bottom`.

  - Interesting sprites and many other things can be done with PlantUML. Read the docs.

This particular workflow is inspired by Bernt Hansen –while quickly searching through the PlantUML manual: The above is known as an "activity diagram" and it's covered in §4.

5.8 Jumping without hassle

```
(defun org-goto-line (line)
  "Go to the indicated line, unfolding the parent Org header.

   Implementation: Go to the line, then look at the 1st previous
   org header, now we can unfold it whence we do so, then we go
   back to the line we want to be at.
  "
  (interactive)
  (goto-line line)
  (org-previous-visible-heading 1)
  (org-cycle)
  (goto-line line)
)
```

5.9 Folding within a subtree

```
; https://orgmode.org/manual/Structure-editing.html
; (describe-symbol 'save-excursion)
;
(defun org-fold-current-subtree-anywhere-in-it ()
  "Hide the current heading, while being anywhere inside it."
  (interactive)
  (save-excursion
    (org-narrow-to-subtree)
    (org-shifttab)
    (widen))
)

;; FIXME: Make this buffer specfic!
(global-set-key (kbd "C-c C-h") 'org-fold-current-subtree-anywhere-in-it)
```

5.10 Making then opening html's from org's

```
(cl-defun my/org-html-export-to-html (&optional (filename (buffer-name)))
  "Produce an HTML from the given `filename', or otherwise current buffer,
   then open it in my default brower.
  "
 (interactive)
 (org-html-export-to-html)
```

```
 (let ((it (concat (file-name-sans-extension buffer-file-name) ".html")))
   (browse-url it)
   (message (concat it " has been opened in Chromium."))
   'success ;; otherwise we obtain a "compiler error".
 )
)
```

5.11 Making then opening pdf's from org's

```
(cl-defun my/org-latex-export-to-pdf (&optional (filename (buffer-name)))
  "Produce a PDF from the given ‘filename’, or otherwise current buffer,
   then open it in my default viewer.
  "
 (interactive)
 (org-latex-export-to-pdf)
 (let ((it (concat (file-name-sans-extension filename) ".pdf")))
   (eshell-command (concat "open " it  " & "))
   (message (concat it " has been opened in your PDF viewer."))
   'success ;; otherwise we obtain a "compiler error".
)
```

5.12 Interpret the Haskell source blocks in a file

```
(defvar *current-module* "NoModuleNameSpecified"
  "The name of the module, file, that source blocks are
   currently being tangled to.

   This technique is insipired by ‘‘Interactive Way to C’’;
   see https://alhassy.github.io/InteractiveWayToC/.
  ")

(defun current-module ()
  "Returns the current module under focus."
  *current-module*)

(defun set-module (name)
   "Set the name of the module currently under focus.

    Usage: When a module is declared, i.e., a new file has begun,
    then that source blocks header should be ‘‘:tangle (set-module ’’name-here’’)’’.
    succeeding source blocks now inherit this name and so are tangled
    to the same module file. How? By placing the following line at the top
    of your Org file: ‘‘‘#+PROPERTY: header-args :tangle (current-module))’.

    This technique structures ‘‘Interactive Way to C’’.
   "
   (setq *current-module* name)
)

(cl-defun my/org-run-haskell (&optional target (filename (buffer-name)))
  "Tangle Haskell source blocks of given ‘filename’, or otherwise current buffer,
   and load the resulting ‘target’ file into a ghci buffer.

   If no name is provided for the ‘target’ file that is generated from the
```

```
   tangeling process, it is assumed to be the buffer's name with a 'hs' extension.

   Note that this only loads the blocks tangled to 'target'.

   For example, file 'X.org' may have haskell blocks that tangle to files
   'X.hs', 'Y.hs' and 'Z.hs'. If no target name is supplied, we tangle all blocks
   but only load 'X.hs' into the ghci buffer. A helpful technique to load the
   last, bottom most, defined haskell module, is to have the module declaration's
   source block be ':tangle (setq CODE "Y.hs")', for example; then the following
   code blocks will inherit this location provided our Org file has at the top
   '#+PROPERTY: header-args :tangle (current-module))'.
   Finally, our 'compile-command' suffices to be '(my/org-run-haskell CODE)'.

   This technique structures "Interactive Way to C".
  "
   (let* ((it  (if target target (concat (file-name-sans-extension filename) ".hs")))
          (buf (concat "*GHCI* " it)))

     (toggle kill-buffer-query-functions nil (ignore-errors (kill-buffer buf)))
     (org-babel-tangle it "haskell")
     (async-shell-command (concat "ghci " it) buf)
     (switch-to-buffer-other-window buf)
     (end-of-buffer)
   )
)

;; Set this as the 'compile-command' in 'Local Variables', for example.
```

5.13 Minted

Execute the following for bib ref as well as minted Org-mode uses the Minted package for source code highlighting in PDF/LaTeX –which in turn requires the pygmentize system tool.

```
(setq org-latex-listings 'minted
      org-latex-packages-alist '(("" "minted"))
      org-latex-pdf-process
      '("pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "biber %b"
        "pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f"
        "pdflatex -shell-escape -interaction nonstopmode -output-directory %o %f")
)
```

For faster pdf generation, may consider invoking:

```
(setq org-latex-pdf-process
      '("pdflatex -interaction nonstopmode -output-directory %o %f"))
```

6 Summary of Utilities Provided

Command	Action
`C-c C-m`	recompile file
`<f5>`	revert buffer
`M-x k`	kill to start of line
`C-`	toggle 2 windows from horizontal to vertical view
`(file-as-list pathHere)`	construe a file as a list of lines
`(file-as-string pathHere)`	construe a file as a string
`(re-replace-in-file file regex whatDo)`	perform an in-file regular expression rewrite
`(mapsto this that)`	regex rewrite in current buffer: this that
`M-x create-scratch-buffer`	–self evident–
`M-x kill-other-buffers`	–self evident–
`M-$`	check spelling of word at point
`M-#`	thesaurus look-up word at point
`(toggle name val )`	*Effectfully* set `name` to `val` only for scope .
`(my/org-run-haskell &optional file)`	Interpret the Haskell org-blocks from a file into ghci.
`C-+/-`	increase/decrease text size
`M-x my-org-html-export-to-html`	make then open html from an org file
`C-c C-c`	execute code in an org `src` block
`<E`	produce an emacs-lisp `src` block
`<g`	produce a graph template `src` block
`C-x t`	open a new untitled org template file
`(org-keywords)`	get `#+Property:  Value` pairs from an org file
`(org-keyword property)`	get the `value` of a given org `#+property`

Since I'm using `use-package`, I can invoke `M-x describe-personal-keybindings` to see what key bindings I've defined. Since not all my bindings are via `use-package`, it does not yet cover all of my bindings.

Some possibly interesting reads:

- Toon's Literate Dotfiles
- Awesome Emacs: A community driven list of useful Emacs packages, libraries and others.
- A list of people's nice emacs config files
- zzamboni's configuration file with commentary
- Karl Voit's article My Emacs Configuration In Org-mode; his init file can be found here.
- Holger Schuri's article Efficient Emacs .org .el tangling ; his init file can be found here.
- Arnaud Legrand's article Emacs init file written in org-mode
- Stackexchange: Using org-mode to structure config files
- A tutorial on evaluating code within `src` blocks