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

Musa Al-hassy

2018-07-25

Contents

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.
Production of org-mode ready-to-go skeletons.
Utilities for working with org-mode files, namely `#+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.

Edit: Recently I switched to spacemacs. I know nothing about spacemacs, I'm just trying it out now and so something below are likely to be super wrong –but they work for me, for now.

1 Introduction

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 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:

```
(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.

```
;; In-case I forget to byte-compile!
(byte-compile-file "~/.emacs")

;; Change this silly counter to visualy notice a change. When making many changes.
;; (progn (message "Init.org contents loaded! Counter: 7") (sleep-for 3))
```

Recently I've switched to spacemacs, and to avoid too much migration I've simply called this literate configuration from within `~.spacemacs` via the `dotspacemacs/user-config` method.

```
(defun dotspacemacs/user-config ()
  "Configuration function for user code.
   This function is called at the very end of Spacemacs initialization after
   layers configuration.
   This is the place where most of your configurations should be done. Unless it is
   explicitly specified that a variable should be set before a package is loaded,
   you should place your code here."

   (org-babel-load-file "~/dotfiles/.emacs.d/init.org")
)
```

Moreover, in the `dotspacemacs/layers` method in `.spacemacs`, I have the following so that spacemacs does not delete layer-orphan packages in an attempt to 'clean up' my unsued packages.

```
dotspacemacs-additional-packages '(htmlize biblio magit haskell-mode dash s)
```

3 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
```

4 Magit & Switching to a new OS

4.1 `use-package`

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 pacakges through `use-package` when it detects they're missing.

The `:ensure` installs the files using `package.el` if need be.

List of additional packages that will be installed wihout being wrapped in a layer. (describe-symbol 'dotspacemacs-additional-packages)

```
(setq dotspacemacs-additional-packages '(htmlize biblio magit haskell-mode dash s))

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

(use-package htmlize :ensure)     ;; Org produced htmls are coloured.
(use-package biblio  :ensure)     ;; Quick BibTeX references, sometimes.

(use-package magit   :ensure)     ;; Efficient version control.
(global-set-key (kbd "C-x g") 'magit-status)

(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`.

4.2 Magit

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

```
(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

(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-box "This file has uncommited changes!")
      (magit-status))))

(add-hook 'find-file-hook 'my/magit-check-file-and-popup)
```

Let's try this out.

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

5 Loads

5.1 `package-initialize`: Melpa, gnu, and org

- **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`

```
(require 'package)
(setq package-archives
      '(("melpa" . "https://melpa.org/packages/")
        ("gnu" . "https://elpa.gnu.org/packages/")
        ("org" . "http://orgmode.org/elpa/")))
(package-initialize)
```

5.2 Programming Language Supports

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 "agda-mode locate")))
```

Sometimes I use Coq,

```
;; Open .v files with Proof General's Coq mode
;; (load "~/.emacs.d/lisp/PG/generic/proof-site")
```

5.3 Unicode Input via Agda Input

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`.

6 Cosmetics

```
;; (load-theme 'spacemacs-dark)
(load-theme 'spacemacs-light)
```

6.1 Column Marker

(Maybe a hook would be better? Much better. . . ?)

Have a thin line to the right to ensure I don't write "off the page".

```
(use-package fill-column-indicator :ensure)
(define-globalized-minor-mode my-fci-global-mode fci-mode
  (lambda () (set-fill-column 90) (fci-mode 't)
))
(my-fci-global-mode 1)
```

There are issues with making things global. In this case, exporting to html produces curious symbols thereby prompting `my-org-html-export-to-html` below to take care of this.

6.2 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
```

6.3 My TODO list: The initial buffer when Emacs opens up

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

(progn (split-window-right)                          ;; C-x 3
(other-window 1)                                    ;; C-x 0
(find-file "~/Dropbox/todo.org"))
```

6.4 Showing date, time, and battery life

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

6.5 Increase/decrease text size

```
(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
```

6.6 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)
```

7 Helpful Functions & Shortcuts

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

7.1 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")
```

7.2 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`.

7.3 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)) )
```

7.4 `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)))
```

7.5 `kill-other-buffers`

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

7.6 `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))
```

7.7 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)
```

7.8 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)
  "Perform the substitution `code[variable  value]'.
   In particular, the value of `variable', if any, *is* unaffected.

   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_)
  )
)
```

7.9 `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")))
```

7.9.1 `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)
  )
)
```

7.10 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.

8 Spelling

I would like to check spelling by default.

`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 "aspell")

;; Maybe a hook is better? Much better ...?
(define-globalized-minor-mode my-flyspell-global-mode flyspell-mode
  (lambda ()
```

```
    ;; spawns an ispell process
    (flyspell-mode 1)

))
(my-flyspell-global-mode 1)

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

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)))))
```

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)
```

9 Org-mode related things

Here is useful Org-Mode Table Editing Cheatsheet.

9.1 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

- `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)
```

9.2 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)
```

9.2.1 `<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"))
```

9.2.2 `<ag` (Org) Agda source template

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

9.2.3 `<hs` Haskell source template

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

9.2.4 `<ic` Interactive Way to C source template

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

9.2.5 `<ich` Interactive Way to C header template

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

9.2.6 `<ver` Verbatim template

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

9.3 `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))
```

9.4 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)
```

9.5 org-mode header generation

Generate an untitled org-mode skeleton file `C-x t` –similar to `C-x C-f` for finding files.

First the template,

```
#+TITLE: ???
#+DATE: thedate
#+DESCRIPTION: A new radical entry of things I'm learning!
#+AUTHOR: Musa Al-hassy
#+EMAIL: alhassy@gmail.com
#+IMAGE: ../assets/img/rwh-200.jpg
#+CATEGORIES: ExampleTags Elisp Haskell Frama-C Specfications Krakatoa
#+OPTIONS: toc:nil html-postamble:nil
,# Other possible are num:nil todo:nil pri:nil tags:nil ^:nil
#+STARTUP: indent

* Abstract       :ignore:
#+BEGIN_CENTER
*Abstract*

This article serves to accomplish *???*.
Write your goal then attempt to realise it, otherwise there's no explicit direction!

#+END_CENTER
```

```
* Introduction

Let's recall concepts ~X~ needed to discuss notions $Y$.

* Middle

We're learnin'!

* Conclusion

Yeah! That was some fun stuff!

* COMMENT footer

# Local Variables:
# eval: (setq NAME (file-name-sans-extension (buffer-name)))
# eval: (load-file "AlBasmala.el")
# End:
```

Then the functionality,

```
(defun new-untitled-org-template ()
  "Produce an org-mode file template."
  (interactive)
  (switch-to-buffer (generate-new-buffer "*Untitled*"))
  (insert (file-as-string "~/.emacs.d/template.org"))
  (org-mode)
)

(global-set-key (kbd "C-x t") 'new-untitled-org-template)
```

9.6 Org-mode cosmetics

```
(with-eval-after-load 'org     ;; Need this here so it loads after the spacemacs 'layer'

;; 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

)
```

9.7 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)
)
```

9.8 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)
```

9.9 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".
 )
)
```

9.10 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".
)
```

*

9.11 Interpret the Haskell source blocks in a file

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

   Note that this only loads the blocks tangled to ‘filename’.
  "
   (let* ((it  (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.
;; Possibly with a (svae-buffer) before hand.
```

9.12 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"))
```

10 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`

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
- 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