The care and conservation of computer files (TC3F)

Grant Rettke

<2014-02-24 MON>

Contents

1 How to design a file conservator (HTDFC)

1.1 Audience [1]

Who should be reading this? Possibly...

[1] `http://dictionary.reference.com/cite.html?qh=audience&ia=luna`

- Entire document: Software engineers who want to do the above [2]
  - If you are using this an excuse to learn Lisp, then know that it is really, really simple. Here is how it goes:
    * 3s to download Emacs
    * 3m to learn how to run code
    * 3h to learn the IDE
    * 3d to master the language
- Decisions & Assembly: Existing Emacs users... daily usage and non-trivial customization An interest in using cask and org-mode literate programming are the only thing that would motivate anyone to read this really.

This is an unfunished and experimental document. All successes with it may be attributed to all of the folks who provided these wonderful tools. All failures with it may be attributed to me.

Why should they be reading this?

> The expectations of life depend upon diligence; the mechanic that would perfect his work must first sharpen his tools. [3]

1.2 Cogito ergo sum [4]

What was I thinking?

creation "the act of producing or causing to exist" [5]

conservation "prevention of injury, decay, waste, or loss" [6]

computer files "a file maintained in computer-readable form" [7]

Pursuing these goals, in the manner of literate programming, also serves to better myself, as captured here:

> The expectations of life depend upon diligence; the mechanic that would perfect his work must first sharpen his tools. [8]

1.3 Means [9]

How do most people do it?

File maintainer a person and program responsible for TC3F

COTS edit, VI, Emacs, IntelliJ Idea, Visual Studio [10]

Bespoke custom software [11]

[2] https://en.wikipedia.org/wiki/Software_engineer
[3] http://www.brainyquote.com/quotes/quotes/c/confucius141110.html
[4] https://en.wikipedia.org/wiki/Cogito_ergo_sum
[5] http://dictionary.reference.com/browse/creation
[6] http://dictionary.reference.com/browse/Conservation
[7] http://dictionary.reference.com/browse/computer%20file
[8] https://www.brainyquote.com/quotes/quotes/c/confucius141110.html
[9] http://dictionary.reference.com/cite.html?qh=tools&ia=luna
[10] https://en.wikipedia.org/wiki/Commercial_off-the-shelf
[11] https://en.wikipedia.org/wiki/Custom_software

1.4 Madness [12]

What about their raison detre? [13]

- Wonderful, wonderful stock tooling, 80%
- Can tool-makers build it perfectly for us all? [14]
- Either way:

  Thinking is required.

1.4.1 Goal

What is my measure of success?

To provide a self-suportable environment in which the creation and conservation of computer files may occur with ease

1.5 Decision

GNU Emacs is an extensible, customizable text editor [15]

1.6 Methodology [16]

How will I customize it?

Agile [17]

- Product Backlog
- Sprint Backlog
- Review, Refine, and Reiterate
- COTS libraries
- Capture rationale and reasons along with things I did or didn't do and why
- Includes links to everything

Note: How you break up the initialization of a system like Emacs is mostly personal preference. Although org-mode (Babel) lets you tell a story, I was coming from a pretty structured config file to begin with. In the future, it might be interesting to look at this system from scratch in terms of doing literate programming. Fortunately, it provides that freedom out of the box.

[12] http://dictionary.reference.com/browse/madness
[13] http://dictionary.reference.com/cite.html?qh=raison%20detre&ia=luna
[14] http://www.wisdomandwonder.com/article/509/lambda-the-ultimate-goto
[15] https://www.gnu.org/software/emacs/
[16] http://dictionary.reference.com/cite.html?qh=method&ia=luna
[17] https://en.wikipedia.org/wiki/Agile_software_development

1.6.1 Medium

How will explain what I did?

- Audience-appropriate presentations
- Reproducible research
- Reusable data structures

1.7 Developmental Values

1.7.1 Traits

- ☒ Pleasing user experience
- ☒ Pervasive orthogonality [18]
- ☒ Self-suportable

1.7.2 Coding

- ☒ Completion
- ☒ Debugging
- ☒ Templates

1.7.3 Editing

- ☒ Auto-indenting
- ☒ Binary file editing, hex editor
- ☒ Code folding
- ☒ Code formatting
- ☒ Diff'ing
- ☒ Heavily used languages:
  - ☒ CSS
  - ☒ Elisp
  - ☒ HTML
  - ☒ Graphviz
  - ☒ JSON
  - ☒ JavaScript
  - ☒ Make
  - ☒ Markdown
  - ☒ R

[18] `http://dictionary.reference.com/browse/orthogonal`

  - ☒ Scheme
  - ☒ shell
- ☒ Lightly used languages:
  - ☒ go
  - ☒ OCaml
  - ☒ SML
  - ☒ Clojure
  - ☒ Perl
  - ☒ Python
  - ☒ Ruby
- ☒ Incremental selection
- ☒ LaTeX
- ☒ SEXP Support
- ☒ Spell-checking
- ☒ Structured navigation
- ☒ Syntax highlighting
- ☒ Tab management

1.7.4 Development

- ☒ Build tools: make
- ☒ Copyright notice, analysis, and standards
- ☒ Dependency management
- ☒ Diagramming
- ☒ UML
- ☒ Version control: git, svn, bzr, cvs, rcs
- ☒ Workflow

1.8 Operational Values

1.8.1 Fonts

- ☒ Focus on easily-screen-readable, mono-spaced
- ☒ Research suggests that san-serif fonts are easier to read [19] , [20] , [21] , [22] , [23] , [24]

[19] http://thenextweb.com/dd/2011/03/02/whats-the-most-readable-font-for-the-screen/#!uCcs8
[20] http://www.webpagecontent.com/arc_archive/182/5/
[21] http://www.awaionline.com/2011/10/the-best-fonts-to-use-in-print-online-and-email/
[22] https://tex.stackexchange.com/questions/20149/which-font-is-the-most-comfortable-for-on-screen-viewing
[23] http://river-valley.tv/minion-math-a-new-math-font-family/
[24] http://edutechwiki.unige.ch/en/Font_readability

- ☒ Research suggests that color doesn't matter; only contrast [25] [26] , [27] , [28] , [29] , [30] , [31] , [32]
- ☒ Unicode support is critical

1.8.2 Images

- ☒ Ascii art

1.8.3 Spreadsheet

- ☒ Calculation
- ☒ Data management
- ☒ Import/Export

1.8.4 Files

- ☒ Auto-save & synchronize
- ☒ Encryption
- ☒ File-system/directory management
- ☒ Project structure
- ☒ Search everywhere

1.8.5 Commands

- ☒ Key recording
- ☒ Macros
- ☒ History of all things: files, commands, cursor locations
- ☒ Undo

1.8.6 Publishing

- ☒ Code
- ☒ Multiple formats: HTML, JS, PDF

1.8.7 Terminal

- ☒ Cross-platform shell
- ☒ Games

[25] http://usabilitynews.org/the-effect-of-typeface-on-the-perception-of-email/
[26] http://usabilitynews.org/know-your-typefaces-semantic-differential-presentation-of-40-onscreen-typefaces/
[27] http://typoface.blogspot.com/2009/08/academic-base.html
[28] http://liinwww.ira.uka.de/bibliography/Typesetting/reading.html
[29] http://www.kathymarks.com/archives/2006/11/best_fonts_for_the_web_1.html
[30] http://psychology.wichita.edu/surl/usabilitynews/52/uk_font.htm
[31] http://usabilitynews.org/a-comparison-of-popular-online-fonts-which-size-and-type-is-best/
[32] http://usabilitynews.org/a-comparison-of-popular-online-fonts-which-is-best-and-when/

1.8.8 Remote file access and management

- ☒ SSH
- ☒ SCP

1.8.9 Music

- ☒ LilyPond

1.8.10 Communications

- ☒ IRC

1.9 Observations

- Went stunningly well
- Stopped logging hours spent >100h
- This config was developed organically quite differently from the original idea
- Literate programming allowed an insanely flexible and freeing experience
- This one massive experience for me made a powerful, positive, life-changing impression on me
- My laziness and poor habits were made quite apparent going through the efforts to consider, realize, and support this system
- Before beginning I had no value system about testing this kind of artifact
- If this attribution, which is *only* about Wolfram's MathWorld [33], the you may be interested in this style of programming and literature:

  Created, developed, and nurtured by Eric Weisstein at Wolfram Research

1.10 La trahison des images [34]

Where else has this acronym shown up?

#**cc33ff** bright purple color [35]

Commander, U.S. Third Fleet WWII navy [36]

A spasmogenic fragment a peptide [37]

[33] `http://mathworld.wolfram.com/`
[34] `https://en.wikipedia.org/wiki/The_Treachery_of_Images`
[35] `http://www.color-hex.com/color/cc33ff`
[36] `https://secure.flickr.com/people/c3f/`
[37] `http://books.google.com/books?id=L4CI-qkhuQ8C`

2 On the role of, and the need for, a personal philosophy

2.1 Audience

- Myself
- The scope of my approach is neither scientific nor entirely thought out or even remotely near perfected. Despite that, the show must go on, and I want to keep a record of how and why I have pursued this goal.
- Most of the work performed within this document will be moreso a work of art, and philosophy, then of science.

2.2 Keyboard usage strategies

2.2.1 Background

My personal keyboard layout has evolved quite slowly over the years. Beginning as a begrudging Emacs user, I quickly learned some basic manners and abandoned it. Problem was that I had no good reason to be using Emacs, and so, I failed. Instead of a solution, I saw it as an obstacle. Lesson learned. When I wanted to learn Scheme, Dr Racket worked out just fine. It wasn't until wanting to learn OCaml that I became smitten with our dear Emacs.

My usage was pretty basic, customizing the bare minimum and sticking with the defaults for everything. That approach is quite fine, for whatever point you are at because you are more or less guaranteed excellent documentation on your environment. This was my setup for years and it worked great. The more comfortable you become, the more you change, and the more changes you make to your configuration.

My configuration file grew, and grew, and grew. It had an ad-hoc layout, and soon I even started to forget why, or where, or how. With additionl and hacks and the usual, eventually I turned to literate programming with org-mode. That was and is pure joy, and I've barely scratched the surface. This was a turning point for me. At its simplest, I was then able to do everything that needed to be done in Emacs, and it was then that I started caring a lot more about how my keyboard was set up.

2.2.2 Version 001

The simplest and best place to start is to remap the control key to the center left of the keyboard. On most keyboards, this is directly next to the "A" key. This change alone served me quite well for years and years. It was after years of usage that I got curious about "better ways" to do; and I suppose that is the driving force behind thousands of Emacs packages. The change works well on Windows, Linux, and OSX. Lately I've read a lot of material published by bbatsov [38] and xahlee [39] about their quests to perfect Emacs and there is a lot of discussion about keyboard mappings that go so far as to talk about how to avoid repetitive strain injury [40] due to QWERTY [41]. By "lately", I mean over the course of a few years. That alone will get any computer user interested in really thinking about their mappings and how to make things easier on their wrists.

2.2.3 Version 002

One of the simplest changes discussed is simply to never twist and contort your fingers into order to perform key chords that both the meta key itself and the key with which you are chording. At first blush,

[38] `https://github.com/bbatsov`
[39] `http://xahlee.org/`
[40] `https://en.wikipedia.org/wiki/Repetitive_strain_injury`
[41] `https://en.wikipedia.org/wiki/Keyboard_layout#QWERTY`

his seemed silly to me, but after trying it out for only a day or two, my wrists and hands simply felt less worn out at the end of the day. That was intriguing having such immediate results. All it took was swapping he enter key with one of the meta keys on the bottom row. The lack of balance though qucily became kind of a nuisance.

Reading a range of links about the topic, I settled on a simple goal of having meta key parity on both sides of the keyboard. That does take some work. The experience resulting from the goal set forth, over the course of a few months, morphed into the desire to grow the chording space into something much more manageable, and began a new phase.

2.2.4 α Version 003

In my minds eye, I see the key chording space in some broad, simple divisions, roughly something like:

Emacs built in bindings, most common, documentation exists for all

Packages separately installed, generally play nice with Emacs

Personal my own key-bindings that try to play nice and adhere to the spirit but generally struggle due to lack of remaining name-space control-land

All of the good key-bindings are used up. The "good ones" are easy to use and easy on your hands and fingers. Even worse, sticking close to the native and package bindings results in having just too many chords to make it nice to use anymore (curiously remembering them is *never* an issue). The mental model that I am beginning to develop is quite simply to segregate all personal bindings into a new key-space, conceptually, so I generally know where to put thing and where to find things.

Articles on things like god mode [42] were my first thought on how to tackle this, but on further review it became clear pretty quickly that the best approach for me would be to follow Xah's advice and start using more meta keys.

2.2.5 β Version 003

My desire is to have a pleasant key binding approach that works on all keyboards and supports all meta keys [43] supported by Emacs [44]. To get started I tracked down some examples of what I want to support for work, home, and other:

- A Macbook Pro Retina 15"
- A Thinkpad T42
- A Lenovo W540
- A Dell external USB keyboard
- A HP EliteBook 8570W

(Still not sure how to track down one of these [45]

Staring at these for a while got me thinking about the "perfect" layout and it started to get a bit overwhelming so I set out to reduce the keys for consideration a bit:

- Total keys: 78

[42] `https://github.com/chrisdone/god-mode`
[43] `http://ergoemacs.org/emacs/emacs_hyper_super_keys.html`
[44] `http://ergoemacs.org/emacs/emacs_hyper_super_keys.html`
[45] `https://en.wikipedia.org/wiki/Space-cadet_keyboard`

Yikes. That is a lot to chew on. Thinking about how I really use the keyboard, though, I now that some keys are not up for debate. Here is what I mean

Keys that will remain the unchanged

`F` **keys, 12** I expect them; that is what makes it a computer keyboard!

Alphanumerics, 48 Numbers, letters, Symbols, Space... they are self-evaluating!

Permanent, 1 This may never change. Ever. The power button!

Frequently used, 3 Delete. Tab: for bash completion. Esc.

Arrow keys, 4 leave the alone it is just right. It just feels wrong to remap it. Used in Finder. It stops videos from playing.

Remaining keys: 11. Now is when I start to look at what keys I really, really need, that I can't live with out. All it takes is a simple question: "How often do I actually use that key?". Additionally, because I want key balance for meta keys, I can drop the number down to 7 because 4 of them were listed twice, conceptually at least.

Next step is to look at the Dell keyboard and the T42 laptop to see what keys remain in what order, and where. Following the layout from top left, counter-clockwise, to top right looks like this. This does include keys that I won't re-map, but I want to list them just to get a sense of the location and remind me of how it "normally looks":

> esc tab caps lock return shift shift fn control option command command option left up down right

The Dell:

> esc tab caps lock return shift shift control alt command command option menu control

The T42:

> esc tab caps lock enter shift shift fn control alt alt control left/down/up/right

Taking a peek at the more modern HP laptop and W540 I find 4 keys available on that bottom row, just like on the Mac. This is something to think about. I don't want to design around the past, but at the same time I would like to have the option of things being mapped nicely regardless of the machine and keyboard... it is just more flexible. At the same time I don't want to be trapped in the past... and at the very same time I do not want to be beholden to an external keyboard. Here is what I decided to do...

Assumption there are only 3 usable keys on each side of the bottom row. This will work for perhaps all machines and hardware out there and the decision will be final. Now I need to figure out the plan.

Having had some really good experience with KeyMapper [46] on Windows and KeyRemap4MacBook [47] on OSX I am feeling very confident and adventurous on pursuing an quite aggressive remap that looks like this:

Caps lock control

Shift option (alt, meta)

Command command, pretzel, windows

[46] `https://code.google.com/p/keymapper/`

[47] `https://pqrs.org/macosx/keyremap4macbook/`

Something hyper

That takes care of all of the meta keys but leaves stranded:

- Shift
- Return
- Caps lock

And I'm not sure what to do with:

- Fn

Then I actually tried setting this up, on OSX!
Working through this was quite educational, here is what worked and here is what didn't:

Hyper I never figured out what key ot use for this. Oops. Fn seemed like a godo option until I reminded myself that I like to use the function key for stuff like volume and csreen brightness, so that was out.

Shift when I remapped shift to meta, sometimes it worked right in Emacs and sometimes it just inserted "control" into the buffer. There is an answer, but I chose not to pursue it right now. This alone felt too veery off into a path too far off the mainstream.

Enter it was horrible trying to use the tiny, bottom option key or enter

This has been a good experience and it led to my new/old/new configuration that was basically a slight improvement, that will basically work everywhere, and is in fact not very radical.

2.2.6 Version 003

The story is still simple, yet powerful... the definition of elegance!
First, leave every mapping alone, keep it fresh from scratch and an Emacs and OS perspective.

- Modifier keys:
  - caps lock → control
  - control → caps lock
  - option/alt → option/alt
  - command/windows → command/windows

Second, find a way to make enter key send enter when pressed act as control when held. We really lucked out here, and bbatsov already figured this out for us here [48]. It is kind of cool that many of us will reach the same conclusion as him, and of course also that he graciously blogged the solution. Yet to be done is to find a good solution for Windows and Linux.

Third, super will be provided by option/alt... this is a good choice, as it is used elsewhere for a similar intent, at least in OSX and Windows (windows key). This symmetrical bindings supports quite easy and uniform access to a grand total of 46 keys. That is all with a single key chord! Great to know. Very nice. Doh!... as I never though to inquite about this before.

Fourth, that leaves hyper. Who wants to ditch hyper? I don't. We need a key for it. It would be nice to have symmetry, and by that measure alone I'm not sure where to put it. Fn lives on OSX and Windows keyboards, but I want that. All of the other keys I was looking at have their place and use, and I'm not

[48] `http://batsov.com/articles/2012/12/06/emacs-tip-number-7-remap-return-to-control-in-osx/`

ditching them. That leaves one place, the F keys. F1 and F12 are open. Would ie nice to use them for super? Would it be horrible? Is it even possible? Well, not really. This article [49] explains the notion of reserved keys, and how F1-F4 are not available, thus negating the chance to have balanced hyper on each side using F keys. That is OK. That is sticking with my philosophy of "close to the original" and I feel like it is very OK since we have 46 keys available to find, and bind, however we wish for just "ou

2.2.7 Version 004

1. Beginnings

   Being able to succed, at anything, requires a goal. During the pursuit of the goal, the pursuer changes, and thusly, so does the goal itself. My goal in this section was to capture this iterative process so that I may see how it developed.

   Two ideas had been lingering for me:

   - How to automate key binding configuration and how
   - How to define as simpler, and cleaner philosophy

   The *good* things that keep coming to mind are simple:

   - Stay close to the default bindings,
     - Already know them
     - Documentation is plentiful
     - Others may use
   - Honor the default bindings
     - If possible, never alter them
     - Inform the operator when they *are* changed
   - Honor operator actions
     - Recognize how they use the keyboard
     - Conserve their energy
   - Honor operator preferences
     - Everyone is different
     - Find a general approach that may work for all

   With those values in mind, incluidng all of the exploration that came before it on this topic, providence stepped in.

2. Studies

   Providence, stepped in, kindly, and gently, to point me in the right direction.

   (a) MASTERING KEY BINDINGS IN EMACS
   First, Micky stepped in [50] with a potent summary [51] of where to begin mastering your keybindings. This is *critical*.

   - Grokking `self-insert-command` helps grok the notion of composability

[49] `http://www.masteringemacs.org/articles/2011/02/08/mastering-key-bindings-emacs/`
[50] `http://www.masteringemacs.org/about/`
[51] `http://www.masteringemacs.org/articles/2011/02/08/mastering-key-bindings-emacs/`

- 3 key categories
  undefined key does nothing
  prefix key `C-x` and `C-c`, compose complete keys
  complete key when input, executes
- Some useful key mod commands
  - `define-key`
  - `local-set-key`
  - `global-set-key`
  - `global-unset-key`
  - `local-unset-key`
- Use they `kbd` macro
- Function and navigation keys require angle bracket wrappers
- `remap` thoroughly replaces existin bindings
- Reserved keys
  - In theory, `C-c *` is for you
  - In practice, who knows
  - `F5+`
  - Super
  - Hyper
- Keymap lookup order, first-found, minor modes are first
  - `overriding-terminal-local-map`
  - `overriding-local-map`
  - Inside of char properties [52]
  - `emulation-mode-map-alists`
  - `minor-mode-overriding-map-alist`
  - `minor-mode-map-alist`
  - Inside of text properties [53]
  - `current-local-map`
  - `current-global-map`
- *commands* are *interactive functions*
- Key bindings may only invoke *commands* with no parameters
- `repeat-complex-command` is something that anyone who performs automation may love

Whether the topics are old news to you or new and fresh, that is a delightful post.

(b) Custom Global Emacs Bindings with Key Chord and the Semi-Colon Key
Justin posted this [54] his approach here, and I think that I understood his goals. His comment that:

> learning Emacs and molding it with lisp is a great creative exercise

is **spot on**.
His advice on how to use key-chord mode [55] also struck a note with me:

[52] `https://www.gnu.org/software/emacs/manual/html_node/elisp/Searching-Keymaps.html`
[53] `https://www.gnu.org/software/emacs/manual/html_node/elisp/Special-Properties.html`
[54] `http://blog.waymondo.com/2013-01-14-custom-global-emacs-bindings-with-key-chord-and-the-semi-colon-key/`
[55] `https://github.com/emacsmirror/key-chord`

> this is basically an empty binding namespace... you can use the most memorable mnemonic letters... You don't have muck around with overriding or conflicting with command prefixes between lisp packages or memorizing multi-command. You also don't have to rely on bindings involving the super key (in OSX), which may conflict with system-level bindings... The biggest consideration to make when defining chords in general is to stray from key combinations you might accidentally fire when typing away....

Justin shared a concise bit of wisdom that is, like most things you will find in this community, a pleasure to consume.

EmacsWiki shared some details [56], too:

- The term *key chord*
  - Is specific to using this mode
  - Is two keys pressed simultaneously
  - Or is a single key pressed twice quickly
- Use the thumb a lot, it is strong!
- Avoid chords common to how you "write"

Of course, Magnar already knew [57], yet further evidence that all of his vlogs are required viewing.

(c) key-chord.el [58]

Many times, the source *is* required-reading, too

- `key-chord-define-global`
- `key-chord-define`
- Everything that I noted in the blog posts is started in the code itself!
- Recommends that chords only involve two fingers on one hand to keep it fast
  - Interesting because I was asserting that two-hands would be fine since I use the control key plenty of times, and it would keep the key space open
  - Wonder how important this one is
- You can't use function, control, or non-English letters
- Only 2 keys are supported!

(d) Emacs: How to Define Keys

This article [59] will serve many:

- One may define bindings of:
  - A single keyA sequence of single keys
  - Key combinations
  - Sequence of single/combo keys
- Keys to Avoid (rebinding)

  Control characters that may be represented by a `C-?`

  `F1` **or** `C:h` they hep!

  `ESC` **or** `C:[` complicated meanings

  `C:up S:letter]` doesn't work in terminals

  `C:m` **or** `Enter` they are linked

[56] `http://www.emacswiki.org/emacs/KeyChord`
[57] `http://emacsrocks.com/e07.html`
[58] `http://www.emacswiki.org/emacs/key-chord.el`
[59] `http://ergoemacs.org/emacs/keyboard_shortcuts.html`

`C:i` **or** `TAB` they are linked

- There are so many keys that if you define your own then you are probaby doing it wrong
- Good Key Choices
  - Someone else has thought through all of this, *too*, then!

Always good	F5-F9, F11-F12
Usually good	F1-F4, F10
Excellent (check OS use)	C:F1 - C:F12
Excellent (check OS use)	M:F1 - M:F12
Excellent (check OS use)	S:F1 - S:F12
n't use `digit-argument`)	C-# and M-#
and page navigation keys	maybe
Super and hyper	all good

- So that is how you may enter diacritics

(e) A Curious Look at GNU Emacs's 1000+ Default Keybinding

Something of a diversion for me having covered stuff elsewhere, but this [60] is a nice to know, too:

- The fact that `F1` is bound to help really reveals how thoughtful Emacs was provided for its users
- Special symbols are nice to know about, I've always used `ucs-insert` and this might be a nicer option, even for guillemots.
- F
  - 3 starts a macro recording
  - 4 ends or runs it
  - 10 opens the menu bar

(f) Emacs Keybinding: the Power of Key Sequences

More [61] from Xah's great pool of Emacs wisdom. Something great to think about when you design your layout:

- Yet another keyspace
- Use these when there is discontinuity in your in your editing
- Choose F keys when you need a break

3. Discussion

Wow. Learned so much. Thought a lot, too. Xah's list of *god keys* alone would be enough of a place to finish because it opens up the key-space so much without requiring anything more than the *default* Emacs setup. Sticking wiht that is really kind of intriguing, but for the fact that you are still *always* going to be using meta keys. Defining your own key sequences, though, opens the door for faster approaches. Combinding the two sounds interesting, too. *This* is where key-chord starts to get really interesting in the sense that it opens up new venures for thoughtfully choosing, or defining, a sort of meta-key, any-how that you wish. Very cool.

4. Assumptions

The key-chor package will *just work* in the same manner that every other built-in Emacs feature *just works*.

[60] `http://ergoemacs.org/emacs/gnu_emacs_keybinding.html`
[61] `http://ergoemacs.org/emacs/emacs_keybinding_power_of_keys_sequence.html`

5. The plan

   - Tenets
     - 99.999% of the time, leave *stock bindings* alone
     - Never use
       * `C-c`
       * `C-x`
       * `F1 - F-4`

Home key chords asdf-→ or jkl;← gives 12x8=96 bindinds!

· Upon thinking this through, decided not worth the trouble because typing quickly would trigger the chords

     - Initially, focus on global mappings to "keep it easy"
   - Proximity
     - **CLOSE** (continuity keys)
       * Use alphanumeric/symbol key-chords when
       * The combination is obvious… like ".." → "…"
       * There is **no** chance that it could inadvertently be pressed; recalling that *both* directions must be considered
       * Excluding alphanumeric leaves; ` ‘ - = [ ] \ ; ’ , .  / `
       * Likely offenders: `- = ; ’ , .`
       * Leaving: ` ‘ [ ] \ / `
       * Left hand: 12x5
       * Right hand only: 7x5
       * 95 possible if this is right
       * Preferences: [,], \…
       * Likelihood of usage? low
         · Choosing a global mapping that works in all modes will involve a lot of work
         · Tough to choose these when `META` keys are easily within reach instead
     - **NEAR**
       * Goal is to minimize finger travel
       * Keys used most frequently
       * In order of preference:

C-#keys 10

C-Fkeys 6 (3 easily in reach on one hand)

     - **FAR** (discontinuity keys)
       Super:any-one (12+10+11+26)=59
     - **FURTHER**
       Use `[S|C|M|S]F:5+` keys 8x4=32 bindings

200+ bindings (stated C-Fkeys extra for easy reachers), that is fine.

6. The result

   (a) Preference in terms of frequent use, is ease of use:

(b) `C-#`

(c) `M-#`

(d) `C-F`

(e) `M-F`

(f) In the process, realized that I didn't consider using upper case letters to

chord with! One may argue that defeats the purpose. I argue that it makes S serve as a quasi `META` key and that is fine because for some reason, mentally, it makes more sense, and feels a bit different from a `META` key which has a slightly different intent.

(a) It might involve re-training the operator, because unlikely things, like

typing in all caps, can muck with desires for using key-chords like `META`.

(a) Chords exactly right next to each other are definitely just nice and pleasnt

to press.

(a) The ease of access becomes quite clear. For me it is `C-[123]` and C-90[-] that

are quite easy to reach. Good to know and note, note sure how though yet.

(a) Choosing a place to map from, in terms of the distraction level, kis kind of

an odd experience. I had ideas about how the "disruption level" would be all that was needed to figure out where/how to provide a mapping. That was mostly true, and is still the case, but I just had a surprise where once I learned about how useful ace-jump is *all the time*, I realized that I must somehow have a home key chord. Previously, I had determined that there is no point because of the assumptions to bother using English language keys. Staring at the keyboard though, with this new understanding of the power of this mode, it became really simple: `d` and `f` (or `f` and `d`) are rarely if ever used together, so clearly that is the right place for a key-chord for ace-jump!

(a) My ideas, some were good, some were not. Interesting to see how they pan out

and develop in the log here. It almost does seem like anything **not** involving vowels may be a good candicate for a key-chord. I initially just didn't want to have to think about it, especially during an active touch-typing spoken-language even. Whatever happens, I'm trying to keep an open mind and let it develop organically.

(a) Realized today that since I use vc-next and er/expand-region **so** much that

they should be even **closer** to home. Fixed that.

(a) Looking at the version control mappings to s-d... that was a relatively easy key combo, and naturally it ended up with the choices all occurring on the right hand. That was almost without thinking.

(b) Another idea of name-spacing is to use sequences of letters that are meaninful. This [62] articles covers something that I never though to do: C-c word. Simple. Instead of limiting it to meta sequences, like C-x C-e, just do C-x ce!

(c) Just added a key-chord for =a'= because I do that **all the time**. Left-control and ' are already used, and this just popped into my mind. Perhaps it is obvious?

[62] `https://aaronhawley.livejournal.com/29311.html`

2.3 Ponderings

2.3.1 Make things "secure by default" [63]

Your artifacts may end up in use anywhere by anyone. Create an environment where the default configuration is also the most secure configuration. For example, provide HTTPS links over HTTP, and think about what code may run and what it may do. This approach, while admittedly valuable, is inherently at odds with the fundamental mission of a software developer: to enable. Respect both sides of the coin, and you will suffer less.

2.3.2 What it means to test

Testing is like flossing, everybody knows why it is important, agrees that it is important, and even wants to do it... yet does not. Your job is to create an environment where people want to test. The first step is to define a measure of success.

In order to succeed, one must have a measure. Although arbitrary, measures must be made. Think through the problem, the original goals, and the newly understood constraints. All of those things will define the measure of success.

Guided by that measure (or constraint), then you may go about creating an environment that is pleasant and facilitate the achievement of those goals. Practicies like breathing practices and meditation will serve one well here.

2.3.3 Practice

The old tenet that practice makes perfect couldn't be any more true here. In the cycle of learning, you learn the tools, the problem, then apply them, and then, the tools and te problem change you, and the cycle repeats. Lisp programmers who have invested in code-generation (macros) know this well, and yet are constanty surprised when it occurs yet again. In the same style, working with literate programs grows and blossoms in unexpected directions. Though some are painful and irritating; the common thread among all of them is that they all lead somewhere wonderful. With time and practice, you will find yourself not only maintaining things you never intended, but simultaneously pondering and realizing things you had never intended, either. That trip is delightful.

2.3.4 Audience

A question that every document author must both ask themselves, and consequently answer, "for whom am I creating this document?". At one's day job, it is easy: the stakeholder. In our personal life though, most of use don't get into habit of viewing ourselves as the stakeholder, or our family and friends, either. We would be well served to do so, though. One's personal life is a safe, non-trivial place to learn how to better ourselves. Starting with onesself is a perfect place to start. That is not to say that starting with others isn't also great, it just wasn't the right place for me.

Originally I wanted to create a document explaining to others my goal for this document. That was a honest yet ultimately misguided effort, because I didn't know where it was going. At the beginning, I had a very different belief system. My intention of focusing on those things was good, it just needed to be simplifed and re-focused upon doing that work for myself. At delivery time, I am the single stakeholder and all efforts should be focused there. Where is there?

There is in me, a flawed, irrational, and illogical human being. Quite common, actually, but we still like them. The document that I must deliver has to account for all of those things, and help me to achieve my goals.

[63] `https://www.openbsd.org/security.html`

2.4 Philosophy

A favorite fable is that of the human who upon reaching the afterlife, meeting his Holiness, vented his frustration exclaiming "All this time... for my whole life I begged to you that I wanted to win the lottery! And you, you never let me win, it would have made my life so, so much better. You failed me". With a kind heart and a sweet voice, his Holiness explained "My child, I did let you win, you just never bothered to get off you ass to go and purchase the ticket". That is certainly a favorite of mine, it captures a specific idea quite succinctly and humorously: that effort is required.

These days, at least here in America, a land of great, great properity, the most propsperous people have grown lazy and selfish. A sense of entitlement abounds, surrounds, and consumes them. This disposition reveals itself in every action that they take ranging from beliefs on public policy and whether or not to donate money to the needy all they way down to their day jobs.

Nearly all of the great technologies on which the modern world is built, at least from a software perspective, occurred due to the efforts of great inviduals, who may be broadly and perhaps unfairly lumped under the singular umbrella of The Free Software Foundation [64].

Sadly most computer professionals today, especially developers, make a non-trivial percentage of their income using the artifacts produced by the combined efforts of others, yet give little to nothing back to that community. From the simplest form of contributing money, all the way down the cheapest form of simply promoting its values, most people are too lazy and selfish. Honestly, I understand though, I used to be that way, too. The important thing is that it is never too late too change.

Our mind is here to be used; fight laziness. Our efforts are here, to help contribute to and serve others; alwys give back to your community no matter what it may be.

2.5 The desire

"I want". If only all conversations would start out with a clear goal in mind. All too often we waste our own, and other people's time talking and simply trying to figure out what it is what we want. For most of us, "it", is that thing that will solve all of our problems in life and make us happy. Technology is no exception.

The perfect integrated development environment is a topic of constant conversation. For good reason, for most of us it is our only tool. Unlike carpenters and wood-workers who have a bevy of interesting and delightful tools, we are stuck with but one. Fortunately for us, our singular tool allows limitless creation, of tools and more. Alan Kay said it so well [65]:

> The computer is a medium that can dynamically simulate the details of any other medium, including media that cannot exist physically. It is not a tool, although it can act like many tools. The computer is the first metamedium, and as such it has degrees of freedom for representation and expression never before encountered and as yet barely investigated. The protean nature of the computer is such that it can act like a machine or like a language to be shaped and exploited.

Even more succinctly, my measure of success is to:

To provide a self-suportable environment in which the creation and conservation of computer files may occur with ease

As of writing, although there are many nice options out there, none of them come within even light-years, of power that you are granted for working with a computer as that metamedium, that GNU Emacs [66]. With that in mind, the following is what I actually want to do with it.

[64] `https://www.fsf.org/`
[65] `https://www.cs.indiana.edu/~rpjames/`
[66] `https://www.gnu.org/software/emacs/`

2.6 The preparation

> Give me six hours to chop down a tree and I will spend the first four sharpening the axe. [67]

Even better, configure Emacs properly and you will end up with a lightsaber. It takes investment though, and it begins with preparation. For me, that meant getting some real life experience, learning new things, getting unpleasant phone calls when systems went down, and perhaps most importantly forcing myself outside my comfort zone.

We work so hard to become experts, yet as a result of it, we close our eyes to new possibility and techniques and approaches, that when combined with our existing experience, could help us to produce some really beautiful things. That experience is often reflected in the love, adoration, and resect held collectively for the Lambda papers [68].

At its simplest, reading about Emacs and org-mode are a perfectly fine place to start with this kind of a document.

2.7 Expressivity

Words are our fundamental form of persistent communication. Images and music are quite delightful for other kinds of communication, but usually not for data. At this point, Unicode is the best option for symbolic representation of ideas, and its use should be embraced, and expected by all programmers.

2.8 The story

The creativity that you apply and capture to assemble your system... this is where all of the fun stuff is. Let me elaborate, everything in your artifacts are valuable because they tell the story. Actually, they tell the story about a story, a story that has yet to occur and also a story that has previously occurred. It is here, where the actions lives, that all of those things are learned, practiced, suffered accordingly from, and reveled in! In other words, it is yet another story, a fun one.

If you haven't noticed by now, either by hearing rumors, reading accounts, or learning of it yourself: human beings are story-oriented. Your ability to successfully function in and contribute to society will be directly proportional to your ability to listen to stories, tell others' stories, live your life such that you have new stories to tell, and capture them in some form of persistent storage. Stories grant us the power to learn from others wisdom that was painfully acquired thousands of years ago, and it gives you a chance to contribute the results of your hard work, for the future of humanity, too. A belief system about the value of story-telling is essential, critical, and mandatory to successfully achieve your goals with literate programming.

As I change, the story will change, and the action will change. The cycle will never end.

Nevertheless, I will attempt to do my best here with the good part of me being a flawless, rational, and logical human being to:

- Deliver a supportable system
- Deliver an adaptable system
- Deliver an expandable system

2.9 Inspirations

Eric Weisstein Creator of MathWorld [69]

[67] `http://www.brainyquote.com/quotes/quotes/a/abrahamlin109275.html`
[68] `http://library.readscheme.org/page1.html`
[69] `http://mathworld.wolfram.com/about/author.html`

2.10 Reminders

- **NEVER** edit source blocks outside of their editor mode
  - **Guaranteed** issues will occur if it is LISP
- Treat source blocks amazingly delicately and thoughtfully because if you don't then you will break your system
- The flow is
  - First make the new changes directly in the code
  - Verify that they work
  - **Then** place them in this document
- Only use in-line footnotes unless your document is very very small
  - Footnotes in org-mode are really, really great. Before you really get into using them, take a bit of time to think about how you want to use them.
  - If you have 5 footnotes or less, then don't think anymore about it. If more then read on.
  - This topic is not unique to org first of all, it just isn't something that you consider much until it is too late. Once you get into the org lifecycle, you start tossing and slinging document and code fragments with ease, especially while refactoring. This is all find and well, until you realize that your footnotes will be left sad and alone, abandoned for some cruel fate. In particular, it will break your document.
  - The better way is to define them all in-line; that will allow simple and easy refactoring in a quite pleasant manner.
- No comments in generated source code ever; barring a few special cases.

3 Decisions

Given values and restrictions, review, identify, and evaluate available options.

Eventually I realided that the system itself needs to be self-suportable. In other words, stage the user for success by either leaving the system in a runnable and usable state or notify her when something is not happy. This lesson was learned when I spent a few hours etting up erc and somehow got the bizarre idea that I would remember where all support files belonged (or was it dired?). Big mistake, we don't have to remember things that computers remember for us (or rather persistent memory). This document is consequently set up in a manner that will provide a self-supportable user experience at nearly every level possible.

3.1 General stuff [70] , [71] , [72] , [73]

A number of variables are generally important, and are also general. Yet again my failure to RTFM has taught me a lessson; `setq-defaul` is for buffer local variables and `setq` is for global variables. As the manual points out, you probably only want to be doing the former in an init file. Be sure to read all the

[70] `https://www.gnu.org/software/emacs/manual/html_node/emacs/General-Variables.html`
[71] `https://www.gnu.org/software/emacs/manual/html_node/elisp/User-Identification.html`
[72] `https://www.gnu.org/software/emacs/manual/html_node/emacs/Init-Examples.html`
[73] `http://nic.ferrier.me.uk/blog/2012_07/tips-and-tricks-for-emacslisp`

links here as they are all important. To reiterate, the most general and reusable setting should be done in the former, and the setting specific to a particular mode or situation should be done with the latter. At this point I think I understand the intent, but do not yet have a good strategy to follow for when to use them other then a very broad: for stuff that is generally a great setting for 80% of situations, do the former; and stuff that is great 80% but only for a specific mode for example, do thel latter.

`boundp` and `fboundp` are useful here, too. Initiially I had thought that general variables were the place to put most stuff, but as their simplest they should remain external and not be managed by my init scripts, so I learn towards the more specific versions here.

```
(setq-default user-full-name "Grant Rettke"
              user-mail-address "gcr@wisdomandwonder.com")

(setq-default eval-expression-print-level nil)
(setq-default case-fold-search nil)
```

Starting to think that I might want a "useful library" section, because I just added the xml-rpc [74] library here because it is clearly useful:

```
;; TODO: Move this to a lib section after Cask (require 'xml-rpc)
```

Another notable note that touches upon why `cons` cells shouldn't be directly manipulated:

(info "(elisp) Rearrangement")

A common theme in the modes and before various operations is to save all buffers. This is a desire, to have all files persisted so that everything run *just works*. I'm not quite sure how to codify and automate this yet, but I am on the path.

Performance, give Emacs more RAM:

```
(setq gc-cons-threshold (* 25 1024 1024))
```

3.2 Environment

On OSX, I learned that when you start the GUI version of emacs that it doesn't inherit the `ENVIRONMENT`. This is the solution.

```
(require 'exec-path-from-shell)
(gcr/on-osx (exec-path-from-shell-initialize))
```

For a while I went on a quest to get the `Message` buffer to include timestamps on each entry. EmacsWiki had some decent approaches but none of them worked right for me and I didn't want to dig further. Eventually though I got tired of having to pay close attention to the minibuffer or `Messages` for stuff and just started looking for GUI options. The plan is to have `Messages` for most stuff and if there are alerts by any definition then I want that to be an option. First choice was todochiku [75] due to the high download count but two issues, it didn't work and it is not used by anything else. Alert [76], on the other hand, is, and also lives on Github meaning that it is maintainable.

```
(require 'alert)
(setq alert-fade-time 10)
(gcr/on-gui
 (gcr/on-osx
   (setq alert-default-style 'growl)))
(setq alert-reveal-idle-time 120)
```

[74] http://melpa.milkbox.net/#/xml-rpc
[75] http://melpa.milkbox.net/#/todochiku
[76] http://melpa.milkbox.net/#/alert

3.3 Time

There are time zones that I do care to know about:

```
(require 'world-time-mode)
```

3.4 Font (Appearance)

The studies cited above indicate that the two major factors that contribute to readabilty of a document are contrast and font-face. Sayre's law [77] however demands that any number of other things are critical to how your IDE looks! That is OK. This section captures some of the basics to getting the system looking how I like it.

This is a san-serif, portable, massively Unicode supported font. You may easily change the font size using `gcr/text-scale-increase` and `gcr/text-scale-decrease`; font information appears in the =*Message=* buffer and also the mini-buffer. The font size will be the same everywhere; as it is easier to work between graphic and console mode with that consistency. You may bypass that using the built in functions. The color theme seems to provide excellent contrast, though I can't decipher what the creator is actually saying about them. For a while I went between the light and dark solarized theme, and finaly accepted that I'm happy with light for documents and dark for programs. That is not scientific, and I'm OK with that. Fortunately you can theme per buffer. Unfortunately, it doesn't quite work perfectly. It wasn't a big deal until it broke org's export to HTML. Since I needed that especially for right now, I decided to stick with the dark theme, as it is more familiar. As of this writing there are no less than 3 packages that provide solarized. After reading their documentation quite closely it came down something relatively simple: face support. Trying to set up help popups to look decent I noticed that `auto-complete` and `popup` looked horrible. Reading through the different versions, there was only one [78] package that provided so many faces that I needed and the others did not so the decision was easy.

Sometimes you don't like how a characters looks, or don't have access to Unicode. In such cases, pretty-mode displays substitutions for certain occurences of flagged strings, for example replacing the world `lambda` with the symbol .

```
(defconst gcr/font-base "DejaVu Sans Mono" "The preferred font name.")
(defvar gcr/font-size 10 "The preferred font size.")
(gcr/on-osx (setq gcr/font-size 17))
(setq solarized-distinct-fringe-background +1)
(setq solarized-high-contrast-mode-line +1)
(setq solarized-use-less-bold +1)
(setq solarized-use-more-italic nil)
(setq solarized-emphasize-indicators nil)
(load-theme 'solarized-dark)
(require 'pretty-mode)
(setq make-pointer-invisible +1)
```

3.5 UXO (Traits, user experience/orthogonality)

3.5.1 Keyboard

The user experience revolving around th keyboard is usually accounted for by the features built-in to Emacs. Along came something radical, though, in the form of key-chord [79].

[77] `https://en.wikipedia.org/wiki/Sayre's_law`
[78] `https://github.com/bbatsov/solarized-emacs`
[79] `http://melpa.milkbox.net/#/key-chord`

```
(require 'key-chord)
(key-chord-mode 1)
;; magic x goes here →
```

3.5.2 Windows [80]

Menu bars are not required. [81]

```
(menu-bar-mode 0)
```

Make it really obvious where the 80th column sits. [82]

```
(setq-default fill-column 80)
```

The cursor should not blink. [83]

```
(blink-cursor-mode 0)
(gcr/on-gui
 (setq-default cursor-type 'box))
(setq x-stretch-cursor 1)
```

Show line numbers everywhere. [84]

```
(global-linum-mode 1)
```

Activate syntax highlighting everywhere. [85]

```
(global-font-lock-mode 1)
```

Visualize parentheses a certain way. [86]

```
(setq blink-matching-paren nil)
(show-paren-mode +1)
(setq show-paren-delay 0)
(setq show-paren-style 'expression)
```

Don't use audible bells, use visual bells. [87]

```
(setq ring-bell-function 'ignore)
(setq visible-bell +1)
```

This post [88] got me thinking that perhaps it was wrong of me to be happy with simply re-positioning all of my windows after their layout gets changed. Probably, I'm just a simple user and never run into this problem, or perhaps my layout is so simple that restoring it is not a big deal. That said, I've been having a nagging feeling about how exactly I plan to utilize ERC now that I've got it set up and simply avoided the topic for a while. Now is the time to address it. Reading more about winner-mode [89] , [90], though, has sort of got me wondering why I never pursued something like this before now.

[80] https://www.gnu.org/software/emacs/manual/html_node/emacs/Windows.html
[81] https://www.gnu.org/software/emacs/manual/html_node/emacs/Menu-Bars.html
[82] http://melpa.milkbox.net/#/fill-column-indicator
[83] https://www.gnu.org/software/emacs/manual/html_node/emacs/Cursor-Display.html
[84] http://git.savannah.gnu.org/cgit/emacs.git/tree/lisp/linum.el?h=emacs-24
[85] https://www.gnu.org/software/emacs/manual/html_node/emacs/Font-Lock.html
[86] https://www.gnu.org/software/emacs/manual/html_node/emacs/Matching.html
[87] https://www.gnu.org/software/emacs/manual/html_node/elisp/Beeping.html
[88] http://www.wisdomandwonder.com/link/8533/avoiding-window-takeover-in-emacs
[89] https://www.gnu.org/software/emacs/manual/html_node/emacs/Window-Convenience.html
[90] http://irreal.org/blog/?p=1557

```
(winner-mode +1)
```

Window navigation isn't something that I do a ton of... but I still want it to be a nice option when I use IRC and want separate windows. ace-window makes this easy:

```
(setq aw-keys '(?a ?s ?d ?f ?g ?h ?j ?k ?l))
```

For some reason, on OSX dialogs don't work and essentially end up locking up Emacs! Here [91] is the solution:

```
(gcr/on-osx
 (defadvice yes-or-no-p (around prevent-dialog activate)
   "Prevent yes-or-no-p from activating a dialog"
   (let ((use-dialog-box nil))
     ad-do-it))

 (defadvice y-or-n-p (around prevent-dialog-yorn activate)
   "Prevent y-or-n-p from activating a dialog"
   (let ((use-dialog-box nil))
     ad-do-it)))
```

Add this [92] to the list of things to maintain your sanity... how to resize windows. My bindings are in the keybindings section.

3.5.3 Frames [93]

Make the title frame something special. [94]

```
(setq frame-title-format '("the ultimate..."))
```

The scroll bars are actually quite nice. Despite that, I don't actually use them, so there they go. [95]

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

The tool bars are not very nice. [96]

```
(tool-bar-mode 0)
```

Browse URLs in a real browser; nothing against W3C. [97]
EWW looks interesting, too [98]

```
(setq browse-url-browser-function 'browse-url-generic)
(gcr/on-gnu/linux (setq browse-url-generic-program "chromium-browser"))
(gcr/on-osx
 (require 'osx-browse)
 (osx-browse-mode 1))
```

[91] https://superuser.com/questions/125569/how-to-fix-emacs-popup-dialogs-on-mac-os-x
[92] https://www.gnu.org/software/emacs/manual/html_node/elisp/Resizing-Windows.html
[93] https://www.gnu.org/software/emacs/manual/html_node/emacs/Frames.html
[94] https://www.gnu.org/software/emacs/manual/html_node/elisp/Display-Feature-Testing.html
[95] https://www.gnu.org/software/emacs/manual/html_node/emacs/Scroll-Bars.html
[96] https://www.gnu.org/software/emacs/manual/html_node/emacs/Tool-Bars.html
[97] https://www.gnu.org/software/emacs/manual/html_node/emacs/Browse_002dURL.html
[98] http://lars.ingebrigtsen.no/2013/06/16/eww/

Let the mousewheel move the cursor in a sane manner. [99]

```
(setq mouse-wheel-scroll-amount '(1 ((shift) . 1)))
(setq mouse-wheel-progressive-speed nil)
```

If possible, use a better popup [100] . Pos-tip should help [101]. Have mixed feelings about this. First, glad it is here, and a lot of packages do use it. Eventually I'll need to set up a larger font. My desire was to have pos-tip use the current theme values, but I couldn't figure out how and the folks online weren't quite sure either... it wasn't worth pusuing and I copied the values straight out of the theme itself.

```
(require 'pos-tip)
(setq pos-tip-foreground-color "#073642")
(setq pos-tip-background-color "#839496")
(gcr/on-windows
 (pos-tip-w32-max-width-height))
```

3.5.4 Buffers [102]

It is nice to have an indicator of the right column that indicates the maximum depth of the line. My favorite package is fill-column-indicator [103]. Its use shows up in almost all of the modes. While working on this build though the export to HTML included junk characters, so I had to disable it, at least in Lispy modes. My final solution to be able to use this package was to generate two Emacs configuration files, one for general use and one just for doing exports.

Keep open files open across sessions. [104]

```
(desktop-save-mode 1)
(setq desktop-restore-eager 10)
```

Automatically save every buffer associated with a file [105]. This is another IntelliJ holdover. The built in auto-save in Emacs wasn't something that I needed, and this does the right thing. There is a bit more though to it, namely because the interval is only 20s I still want/need to be sure that the file is saved *before* doing anything like running code or doing a build As such, before most operations, all buffers with files attached are saved *first*.

```
(require 'real-auto-save)
(setq real-auto-save-interval 15)
```

Make two buffers with the same file name open distinguishable. [106]

```
(require 'uniquify)
(setq uniquify-buffer-name-style 'forward)
```

Support transparent AES encryption of buffers. [107] See also for library paths [108]

[99] https://www.gnu.org/software/emacs/manual/html_node/emacs/Mouse-Commands.html
[100] https://github.com/auto-complete/popup-el
[101] https://github.com/pitkali/pos-tip
[102] https://www.gnu.org/software/emacs/manual/html_node/emacs/Buffers.html#Buffers
[103] https://github.com/alpaker/Fill-Column-Indicator
[104] https://www.gnu.org/software/emacs/manual/html_node/emacs/Saving-Emacs-Sessions.html#Saving-Emacs-Sessions
[105] http://marmalade-repo.org/packages/real-auto-save
[106] https://www.gnu.org/software/emacs/manual/html_node/emacs/Uniquify.html
[107] http://ccrypt.sourceforge.net/#emacs
[108] https://www.gnu.org/software/emacs/manual/html_node/emacs/Lisp-Libraries.html

```
(add-to-list 'load-path "/usr/share/emacs/site-lisp/ccrypt")
(require 'ps-ccrypt "ps-ccrypt.el")
```

With modern VCS, backup files aren't required. [109]

```
(setq backup-inhibited 1)
```

The built in auto save isn't required either because of the above. [110]

```
(setq auto-save-default nil)
```

Ban whitespace at end of lines, globally. [111]

```
(add-hook 'write-file-hooks
          (lambda ()
            (gcr/delete-trailing-whitespace)))
```

The world is so rich with expressivity. Although Unicode may never capture all of the worlds symbols, it comes close. [112] , [113] , [114]

```
(prefer-coding-system 'utf-8)
(gcr/on-gui
 (setq x-select-request-type '(UTF8_STRING COMPOUND_TEXT TEXT STRING)))
```

Emacs has a powerful buffer tracking change system. Unfortunately, I don't understand any of it. Undo should "just work".

```
(require 'undo-tree)
(global-undo-tree-mode 1)
(diminish 'undo-tree-mode)
```

Sometimes it is a problem when you havne't got a newline ending a file with source code before it... org-mode is one such case. Require that evry file have a final newline before saving it.

```
(setq require-final-newline t)
```

For a long time I wanted auto-revert everywhere and for some reason gave up on adding it. What the heck? I am human.

```
(global-auto-revert-mode 1)
```

How to jump to locations in a buffer in an easier way than by using the built in key bindings? Science... that is how.

This package [115] searches for the character for which you are searching at the start of a word, highlights matches, and presents you with the letter to press to jump to the match. You may also search in the middle of words. The key to using this to utilize `pop-mark` to get back to where you were.

[109] https://www.gnu.org/software/emacs/manual/html_node/elisp/Making-Backups.html
[110] https://www.gnu.org/software/emacs/manual/html_node/emacs/Auto-Save-Control.html
[111] https://www.gnu.org/software/emacs/manual/html_node/emacs/Useless-Whitespace.html
[112] https://www.gnu.org/software/emacs/manual/html_node/emacs/International.html#International
[113] https://www.gnu.org/software/emacs/manual/html_node/emacs/Recognize-Coding.html
[114] https://www.gnu.org/software/emacs/manual/html_node/emacs/Output-Coding.html
[115] https://github.com/winterTTr/ace-jump-mode

```
(autoload
  'ace-jump-mode
  "ace-jump-mode"
  "Emacs quick move minor mode"
  t)
```

A long, long time ago I saw a neat feature in Sublime Text [116] (their zeal for their editor is great, very sweet) where you could see a miniature version of your buffer off to the side of the buffer itself. Wasn't totally sure what I would use it for, but it was really neat. Ended up on this [117] page but I didn'tw ant to depend upon CEDET [118]. Then, Sublimity [119] showed up in a post somewhere. The timing was perfect because I was getting really curious about a "quiet mode" that didn't show the modeline or the line numbers or fringe, and I didn't feel like implementing it at that moment. It turns out that this package does it all already; very cool.

```
(require 'sublimity)
(require 'sublimity-scroll)
(require 'sublimity-map)
(require 'sublimity-attractive)
```

By default, the map is hidden while scrolling and this makes it work in a responsive and pleasant manner [120]. It makes Emacs quite slow actually. Unfortuantely, having the map constantly disappear is really unpleasant, and the slow down is, too. Well, this will be a balance. I'll turn it on and live with it. The cool thing here is that you may imagine exactly how this is implemented if you've ever set your font manually and used indirect buffers.

```
(sublimity-map-set-delay nil)
```

Usually you actually need two scratch buffers, one for emacs lisp and one for text:

```
(let ((text-buffer (get-buffer-create "*text*")))
  (with-current-buffer text-buffer
    (text-mode)
    (insert "Shall we play a game?")
    (beginning-of-line)))
```

Navigating a buffer was never slow... until learning about ace-jump-mode [121]. The idea is so deceptively simple, and when you grok it, you will be truly shocked. The author sums it up quite succinctly

```
(autoload
  'ace-jump-mode
  "ace-jump-mode"
  "Emacs quick move minor mode"
  t)
(define-key global-map (kbd "C-0") 'ace-jump-mode)
(autoload
  'ace-jump-mode-pop-mark
  "ace-jump-mode"
```

[116] http://www.sublimetext.com/
[117] http://www.emacswiki.org/emacs/MiniMap
[118] http://cedet.sourceforge.net/
[119] https://github.com/zk-phi/sublimity
[120] ttps://github.com/zk-phi/sublimity/issues/10
[121] https://github.com/winterTTr/ace-jump-mode/wiki/AceJump-FAQ

```
  "Ace jump back:-)"
  t)
(eval-after-load "ace-jump-mode"
  '(ace-jump-mode-enable-mark-sync))
(define-key global-map (kbd "C-x SPC") 'ace-jump-mode-pop-mark)
```

Perhaps an odd topic, but how you handle spaces when performing an interactive search is a choice:

```
(setq isearch-lax-whitespace +1)
(setq isearch-regexp-lax-whitespace +1)
```

A lot of times you write things that involves quoting large chunks from other documents. I'm thinking this is more spur of the moment... like in emails. However, this may occur anywhere I suppose. Perhaps coding is another place? At least when you are not doing LP it would be more likely. This [122] seems like a nice way to make it obvious when I insert quoted text:

```
(require 'boxquote)
```

How you move around lines in a file is configurable. My preference is that if I am on the end of a line, and I go up or down, then I want to go to the end of line on that new line. Specifically, I do not want to account for anything special about the character I am dealing with. This is what most folks would expect:

```
(setq track-eol +1)
(setq line-move-visual nil)
```

3.5.5 Modeline [123]

The modelines is capable of so many things. Though I use it for few, I value it greatly. Even the generic, optional options [124] are nice.

Show the file size.

```
(size-indication-mode)
```

It is nice to see the column number, if you are counting columns (not calories).

```
(column-number-mode 1)
```

It is a pain to look at the clock in the GUI bar.

```
(setq display-time-format "%R %y-%m-%d")
(display-time-mode +1)
```

When you load modes, most of them show up in the minibuffer. After you read their name a few thousand times, you eventually quite forgetting that you loaded them and need a diminished reminder. [125]

```
(require 'diminish)
```

[122] https://github.com/davep/boxquote.el
[123] https://www.gnu.org/software/emacs/manual/html_node/elisp/Mode-Line-Format.html
[124] https://www.gnu.org/software/emacs/manual/html_node/emacs/Optional-Mode-Line.html
[125] http://marmalade-repo.org/packages/diminish

Over time you start to, as which everything else in Emacs, think about configuring it "better". Simple things like the file state indicator [126] is one of the first to jump out at you. In my case I've made some nice changes via the built-in mechanisms. Powerline [127] really got me thinking though just because it is so stunning with the use of XPM [128]. Reading through it though, it would require some real digging in, and the documentation doesn't say much and I wasn't sure that I wanted to pursue that much right now. Simple mode line [129] says all the right things, I like their documentation and am not sure whether or not it knows the right things to highlight, or not. How does it know? Clearly there are many ideas [130] on how to customize the modeline. How may we be sure that they are doing it right and displaying everything that mode expects them to possibly be displaying? Like most things it is just trust, and verify. For now it is easier to stick with the built in, and grow it organically. Perhaps more importantly, I *do* like the built-in modeline style.

Make deleting an entire line work how you may expect [131]

```
(defadvice kill-line (around kill-line-remove-newline activate)
  (let ((kill-whole-line t))
    ad-do-it))
```

3.5.6 Mark and Region [132]

When you start typing and text is selected, replace it with what you are typing, or pasting, or whatever. [133]

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

3.5.7 Minibuffer [134]

You will want to configure this at some point.

Make it easier to answer questions.

```
(fset 'yes-or-no-p 'y-or-n-p)
```

It often displays so much information, even temporarily, that it is nice to give it some room to breath. [135]

```
(setq resize-mini-windows +1)
(setq max-mini-window-height 0.33)
```

Allow recursive commands-in-commands show help me keep track of the levels of recursion.

```
(setq enable-recursive-minibuffers t)
(minibuffer-depth-indicate-mode 1)
```

Handle pasting from the clipboard to the minibuffer:

```
(defun gcr/minibuffer-setup-hook ()
  "Personal setup."
  (local-set-key "ESC y" 'gcr/paste-from-x-clipboard))

(add-hook 'minibuffer-setup-hook 'gcr/minibuffer-setup-hook)
```

[126] http://ergoemacs.org/emacs/modernization_mode_line.html
[127] https://github.com/milkypostman/powerline
[128] https://en.wikipedia.org/wiki/X_PixMap
[129] https://github.com/Bruce-Connor/smart-mode-line/
[130] http://www.emacswiki.org/emacs/ModeLineConfiguration
[131] http://www.wilfred.me.uk/.emacs.d/init.html#sec-3-7
[132] https://www.gnu.org/software/emacs/manual/html_node/emacs/Mark.html#Mark
[133] https://www.gnu.org/software/emacs/manual/html_node/emacs/Using-Region.html
[134] https://www.gnu.org/software/emacs/manual/html_node/emacs/Minibuffer.html
[135] https://www.gnu.org/software/emacs/manual/html_node/emacs/Minibuffer-Edit.html

3.5.8 Filesystem management

Not quite sure where this should go yet. Finder is [136] is just fine, and the curiosuity is still there for an in-Emacs solution. Speedbar [137] and SrSpeedbar [138] look nice, as does [139].

3.5.9 Operation (Keybindings/Keymaps) [140]

This section is entirely defined adhering to the philosopy defined above.

Generally disallow stomping of global keymappings unless it makes sense to me (thank you Stefan Monnier for pointing out that referencing the map variable is all it takes):

```
(defadvice global-set-key (before check-keymapping activate)
  (let* ((key (ad-get-arg 0))
         (new-command (ad-get-arg 1))
         (old-command (lookup-key global-map key)))
    (when
        (and
         old-command
         (not (equal old-command new-command))
         (not (equal old-command 'digit-argument))
         (not (equal old-command 'negative-argument))
         (not (equal old-command 'ns-print-buffer))
         (not (equal old-command 'move-beginning-of-line))
         (not (equal old-command 'execute-extended-command))
         (not (equal new-command 'execute-extended-command))
         (not (equal old-command 'ns-prev-frame))
         (not (equal old-command 'ns-next-frame))
         (not (equal old-command 'mwheel-scroll))
         (not (equal new-command 'diff-hl-mode))
         )
      (warn "Just stomped the global-map binding for %S, replaced %S with %S"
            key old-command new-command))))
```

Enable key-chord'ing:

```
<<uxo-keyboard-decision>>x
```

Enable the `super` key-space:

```
(gcr/on-osx
 (setq mac-control-modifier 'control)
 (setq mac-command-modifier 'meta)
 (setq mac-option-modifier 'super))

(gcr/on-windows
 (setq w32-lwindow-modifier 'super)
 (setq w32-rwindow-modifier 'super))
```

[136] https://en.wikipedia.org/wiki/Finder_(software)
[137] http://www.emacswiki.org/emacs/SpeedBar
[138] http://www.emacswiki.org/emacs/SrSpeedbar
[139] https://github.com/jaypei/emacs-neotree
[140] https://www.gnu.org/software/emacs/manual/html_node/elisp/Keymaps.html#Keymaps

These keybindings are custom for me and I've been using them for so long, and that makes it rig+1ht. There are always opportunities for improvement, though. Recently it dawned on me that it is poor-form to waste 3 function keys on the same task, so I rebound `F1` to different modifiers that I hope are consistent, and free up space, too. My preference is to leave `F5`, `F6, and =F7` generally unbound and available for similar operations that are performed in most programming modes. For example, `F5` will execute code in any Lisp-like environment.

Allow these commands:

```
(put 'upcase-region 'disabled nil)
(put 'downcase-region 'disabled nil)
```

Guide:

NON-DISRUPTIVE hands on home, no finger strain, C-[(2|3)|(9|0)]

SLIGHTLY-DISRUPTIVE hands on home, slight finger strain, C-[1|-]

DISRUPTIVE hands on keyboard, not home, palms are home

VERY-DISRUPTIVE hands on keyboard, not home, palms are moved from home

`M-x` truly is the Emacs command line [141].

Echo keystrokes immediately:

```
(setq echo-keystrokes 0.02)
```

1. NON-DISRUPTIVE : KEY-CHORDS

   (a) CHARACTERS

   Save 3 bytes:

   ```
   (key-chord-define-global "3." 'gcr/insert-ellipsis)
   ```

   German umlauts for a, o, und u:

   ```
   (key-chord-define-global (concat "A" "{") (lambda () (interactive) (insert "ä")))
   (key-chord-define-global (concat "A" "}") (lambda () (interactive) (insert "Ä")))
   (key-chord-define-global (concat "O" "{") (lambda () (interactive) (insert "ö")))
   (key-chord-define-global (concat "O" "}") (lambda () (interactive) (insert "Ö")))
   (key-chord-define-global (concat "U" "{") (lambda () (interactive) (insert "ü")))
   (key-chord-define-global (concat "U" "}") (lambda () (interactive) (insert "Ü")))
   ```

   Arrows, so many:

   ```
   (key-chord-define-global (concat "<" "_") (lambda () (interactive) (insert "←")))
   (key-chord-define-global (concat "_" ">") (lambda () (interactive) (insert "→")))
   ```

   Nice for UML, and French?

   ```
   (key-chord-define-global "<<" (lambda () (interactive) (insert "«")))
   (key-chord-define-global ">>" (lambda () (interactive) (insert "»")))
   ```

   ace-jumping is frequent, too, and this one is perfect all home keys same, dominant strong hand:

   ```
   (key-chord-define-global "jk" 'ace-jump-mode)
   ```

[141] `https://aaronhawley.livejournal.com/28413.html`

ace-window is frequent, is beginning to adhere to a pattern of dual approaches:

```
(key-chord-define-global "nm" 'ace-window)
```

Jumping to lines actually happens a lot. When you look at this layout, remember, or rather consider, that what you jump to, or navigate to, is layered in the sense that the granularity of your actions changes and all actions provided here are helpful for those different situations.

```
(key-chord-define-global "fj" 'goto-line)
```

(b) ACTIONS

Do nearly a IKJL style up/down/left/right arrow key, using using chords, saves a trip.

You might wonder why I would use this when ace-jump-window is available? Good question. Sometimes you know exactly what buffer oyu want, so jump to it, and sometimes you just don't, and that is when you do it this way:

```
(key-chord-define-global "JK" (lambda () (interactive) (other-window 1)))
(key-chord-define-global "KL" (lambda () (interactive) (next-buffer)))
(key-chord-define-global "L:" (lambda () (interactive) (previous-buffer)))
```

2. NON-DISRUPTIVE : KEY-MAPPINGS

(a) ACTIONS

These actions appear in order of importance, and thusly frequency of use

Do the *right thing* for getting to the start of the line!

```
(global-set-key (kbd "C-a") 'beginning-of-line-dwim)
```

I use VC quite frequently. This is easy to reach, and does what must be done:

```
(global-set-key (kbd "C-;") 'vc-next-action)
```

Easily select regions:

```
(global-set-key (kbd "C-'") 'er/expand-region)
```

multiple cursor mode... I kept these together because until I make an image, it would be too confusing to keep them in the non and slightly disruptive sections, as they are clearly that:

```
(global-set-key (kbd "M-9") 'mc/edit-lines)
(global-set-key (kbd "M-0") 'mc/mark-next-like-this)
(global-set-key (kbd "M--") 'mc/mark-all-like-this)
(global-set-key (kbd "M-8") 'mc/mark-previous-like-this)
```

3. SLIGHTLY-DISRUPTIVE : KEY-MAPPINGS

(a) ACTIONS

smex integration points:

```
(global-set-key (kbd "M-x") 'smex)
(global-set-key (kbd "M-X") 'smex-major-mode-commands)
(global-set-key (kbd "C-c C-c M-x") 'execute-extended-command)
```

Pop up help:

```
(global-set-key (kbd "s-p") 'gcr/describe-thing-in-popup)
```

ace-window naviation:

```
(global-set-key (kbd "C--") 'ace-window)
```

auto-completeness

```
(global-set-key (kbd "C-3") 'auto-complete)
```

yas expansion:

```
(global-set-key (kbd "C-4") 'yas/expand)
```

Command and uncomment anything:

```
(global-set-key (kbd "C-5") 'gcr/comment-or-uncomment)
```

Do smart new line inside, indenting given the mode:

```
(global-set-key (kbd "s-<return>") 'gcr/smart-open-line)
```

Anything having to do with version control differences:

```
(global-set-key (kbd "s-d h") 'diff-hl-mode)
(global-set-key (kbd "s-d l") 'vc-diff)
(global-set-key (kbd "s-d u") 'vc-revert)
```

4. DISRUPTIVE : KEY-MAPPINGS

   (a) ACTIONS

   These do get used a lot believe it or not:

```
(global-set-key (kbd "C-7") 'gcr/insert-timestamp)
(global-set-key (kbd "M-7") 'gcr/insert-datestamp)
```

   Make auto-complete easily accessible because sometimes other modes bork it just like yas:

```
(global-set-key (kbd "s-<tab>") 'auto-complete)
```

5. VERY DISRUPTIVE : KEY-MAPPINGS

   (a) ACTIONS

   Manage every font size:

```
(gcr/on-gui
 (global-set-key (kbd "s-<f7>") 'gcr/text-scale-increase)
 (global-set-key (kbd "M-<f7>") 'gcr/text-scale-decrease))
```

   Helper stuff:

```
(global-set-key (kbd "C-<f2>") 'emacs-index-search)
(global-set-key (kbd "S-<f2>") 'elisp-index-search)
(global-set-key (kbd "C-<f3>") 'imenu-anywhere)
```

   Resize the current windows [142]:

```
(global-set-key (kbd "s-<up>") 'enlarge-window)
(global-set-key (kbd "s-<down>") 'shrink-window)
(global-set-key (kbd "s-<right>") 'enlarge-window-horizontally)
(global-set-key (kbd "s-<left>") 'shrink-window-horizontally)
```

[142] http://www.emacswiki.org/emacs/WindowResize

3.6 Primary usage

The purpose of this section is to put some visiblity on the modes, how they are used, and where. After configuring a bunch of modes you may find that you want global defaults, don't be afraid of making such changes (and reverting them too).

3.6.1 Custom variables

```
(custom-set-variables
 ;; custom-set-variables was added by Custom.
 ;; If you edit it by hand, you could mess it up, so be careful.
 ;; Your init file should contain only one such instance.
 ;; If there is more than one, they won't work right.
 '(custom-safe-themes (quote ("8aebf25556399b58091e533e455dd50a6a9cba958cc4ebb0aab175863c25b9a
 '(display-time-world-list (quote (("America/Los_Angeles" "Los_Angeles") ("America/Denver" "De
(custom-set-faces
 ;; custom-set-faces was added by Custom.
 ;; If you edit it by hand, you could mess it up, so be careful.
 ;; Your init file should contain only one such instance.
 ;; If there is more than one, they won't work right.
 )
```

3.6.2 Configuration

1. Intellisense (Auto Completion) [143]

   Can you thrive and profit without auto-completion? Surely. The feature is kind of a comfort blanket for most of us; you will never fail to build a system without it (unless you are using Java, then you need IntelliJ). Still it is quite nice to have popup documentation. Still wanting a nice documentation popup, I think that yet again Purcell and friends make our lives easier.

   Thus far, auto-complete has worked fine. More than a few blogposts do mention company-mode [144], so I read up on it. It seems quite nice, but right now I haven't got a reason to explore it further though.

   Still having some mixed feelings about what engine to use to display the popups. Popup itself is quite easy for me to read since it uses the same font as everything else. That alone makes it perfect. Still, the idea of having real popups is intriguing. Either way, both do work, so I will customize as needed. Until I customize the pos-tip font to make it bigger, though, I will stick with the old-fashioned style.

   ```
   (require 'fuzzy)
   (require 'auto-complete)
   (require 'auto-complete-config)
   (setq ac-quick-help-prefer-pos-tip nil)
   (ac-config-default)
   (setq ac-auto-start nil)
   (ac-set-trigger-key "TAB")
   (diminish 'auto-complete-mode)
   ```

2. Whitespace management [145]

[143] http://cx4a.org/software/auto-complete/
[144] https://company-mode.github.io/
[145] https://www.gnu.org/software/emacs/manual/html_node/emacs/Useless-Whitespace.html

Do you need to see tabs and other control characters? Usually, yes.

```
(require 'whitespace)
(setq whitespace-style '(trailing lines tab-mark))
(setq whitespace-line-column 80)
(global-whitespace-mode 1)
(diminish 'global-whitespace-mode)
(diminish 'whitespace-mode)
```

3. Color visualizing [146]

   Nothing against the multitude of RGB hex value web finder web pages... it is just convenient to have it built right in.

```
(require 'rainbow-mode)
(diminish 'rainbow-mode)
```

4. Templating [147]

   Code complating is nice to have; but the second you install it and learn how to use it, you will never find the need to again. Accept it and move on.

```
(require 'yasnippet)
(let ((yas-snippet-dir (concat (cask-dependency-path gcr/cask-bundle 'yasnippet)
                               "/snippets")))
  (when (not (file-exists-p yas-snippet-dir))
    (warn (concat "Can't seem to find a yas snippert dir where it was expected "
                  "at: " yas-snippet-dir " .")))
  (yas-load-directory yas-snippet-dir))
(diminish 'yas-minor-mode)
(yas-global-mode 1)
```

5. Searching / Finding [148] , [149]

   There are many ways to easily find what you need, for a command, for a file, and this mode seems to be a quite nice way. Something I had been curous about but forgotten and stumbled upon again was vertical ido listing, aod I added that back to see how it goes. My initial reaction was that I had wanted this all along, though the transition from looking left-right to top-down was a little unsettling.

```
(require 'ido)
(require 'flx-ido)
(ido-mode 1)
(require 'ido-hacks nil +1)
(require 'ido-ubiquitous)
(ido-ubiquitous-mode +1)
(setq ido-create-new-buffer 'always)
(flx-ido-mode +1)
(setq ido-use-faces nil)
```

[146] http://elpa.gnu.org/packages/rainbow-mode.html
[147] https://github.com/capitaomorte/yasnippet
[148] http://repo.or.cz/w/emacs.git/blob_plain/HEAD:/lisp/ido.el
[149] https://github.com/lewang/flx

```
(require 'ido-vertical-mode)
(ido-vertical-mode +1)
(setq ido-vertical-define-keys 'C-n-C-p-up-down-left-right)
```

6. Project management [150]

   Not everyone likes projects, but I do. There is no perfect middle ground though, that is until this library came along. It is such a joy to use.

   ```
   (projectile-global-mode 1)
   (diminish 'projectile-mode)
   ```

7. Expression Management [151]

   There are a lot of nice options [152] , [153] , [154] , [155] , [156]. For the longest time, paredit was all that I used, but then I started using Emacs for everyone else besides Lisp and was kind of stymied not having great expression management tools. Smartparens seems to have emerged as king, so here it sits. While I was setting up the new config I set this up last... that was a major mistake. After using a good symbolic expression management tool, you quickly forget the nightmare of having to keep expressions balanced yourself. Sure we did fine with VI... but it is so nice to have the tool do it for you. Remember what Olin Shivers said?

   > I object to doing things that computers can do.

   You get a lot of niceties that you would expect like balanced brackets and since there is a strict mode it acts just like Paredit. Additionally you may wrap selections with pairs, auto-escape strings that occur within other strings, and showing matching pairs (of any supported form). `sp-show-pair-from-inside` is kind of interesting. How it works is that normally when your cursor is to the right of a bracket, then the entire expression is highlighted. My assumption is to make it easy for you to see the scope of the s-exp. When you move forward, to the right of that opening bracket, then that highlight goes away. When you set this flag to non-nil, you get a different behavior where just the bracket is highlighted. Not sure how this would help, but still it is kind of interesting to me because it keep your focus. My use case is that you find an s-exp that you want to edit and start doing it, and in that case I wouldn't use this flag. However, say you had wanted to edit and moved the cursor one char forward and were interrupted. Perhaps you would this kind of highlight so when you come back there is still some indicator. From a user-perspective, it just seemed interesting.

   ```
   (require 'smartparens-config)
   (show-smartparens-global-mode +1)
   (diminish 'smartparens-mode)
   (setq sp-show-pair-from-inside nil)
   ```

8. Remote file access [157]

   TRAMP stands for "Transparent Remote (file) Access, Multiple Protocol". It is really, really beautiful.

[150] http://batsov.com/projectile/
[151] https://github.com/Fuco1/smartparens
[152] http://www.emacswiki.org/emacs/ParEdit
[153] http://www.emacswiki.org/emacs/ElectricPair
[154] https://github.com/rejeep/wrap-region.el
[155] https://code.google.com/p/emacs-textmate/
[156] https://github.com/capitaomorte/autopair
[157] https://www.gnu.org/software/tramp/

```
(setq tramp-default-user "gcr")
(setq tramp-default-method "ssh")
```

9. Selection style [158]

   IntelliJ Idea is yet again to blame for being awesome; even the author of this library suffers, or rather enjoys, this phenomenon. When you make a selection of text you typically want to do it in a smart way, selecting the first logical block, then expanding logically outwards, and so on. It could mean selecting a variable, then its definition statement, and then the entire code block for example. Before now I really never had many uses for the `C-u` universal argument functionality for method calls, but if you pass in a negative value before calling `er/expand-region` it will have the nice feature of reversing its incremental selection.

   ```
   (require 'expand-region)
   ```

10. File-system/directory management [159]

    The last file or filesyste management tool that I used wast Norton Commander [160] and then Midnight Commander [161], but my usage was pretty basic. Beyond those basics, I can do even more, basic stuff, in `bash`. Lately I've wanted something a little more consistent, powerful, and memorable, and that led me here. Dired is a user-interface for working with your filesystem; you select files and directories and then choose what to do with them. The ability to customize what you see is included out of the box, and there are additional helper packages [162], too.

    You can use the usual machinery to work with the files. Highlight a region and operation selections occur for all files in that region. Commands are sheduled, and then executed, upon your command. Files can be viewed in modify or read-only mode, too. There is an idea of =mark-in files, which is to select them and perform operations on the marked files. There are helper methods for most things you can think if like directories or modified-files or whatever, meaning you can use regexen to mark whatever you like however you like. If that suits you, then don't be afraid of using the regular expression builder [163] that is built into Emacs. Bulk marked file operations include additionally copying, deleting, creating hard links to, renaming, modifying the mode, owner, and group information, changing the timestamp, listing the marked files, compressing them, decrypting, verifying and signing, loading or byte compiling them (Lisp files).

    `g` updates the current buffer; `s` orders the listing by alpha or datetime.

    `find-name-dired` beings the results back into Dired, which is nifty.

    Wdired lets you modify files directly via the UI, which is interesting. Image-Dired lets you do just that.

    = creates a new directory. `dired-copy-filename-as-kill` stores the list of files you have selected in the kill ring. `dired-compare-directories` lets you perform all sorts of directory comparisons, a handly tool that you need once in a while but definitely do need.

    ```
    (setq dired-listing-switches "-alh")
    (setq dired-recursive-deletes  +1)
    (require 'dired-details+)
    ```

[158] `https://github.com/magnars/expand-region.el`
[159] `https://www.gnu.org/software/emacs/manual/html_node/emacs/Dired.html`
[160] `https://en.wikipedia.org/wiki/Norton_Commander`
[161] `https://www.midnight-commander.org/`
[162] `http://www.emacswiki.org/DiredDetails`
[163] `https://www.gnu.org/software/emacs/manual/html_node/elisp/Regular-Expressions.html`

```
(setq-default dired-details-hidden-string "")
(defun gcr/dired-mode-hook ()
  "Personal dired customizations."
  (diff-hl-dired-mode)
  (load "dired-x"))
(add-hook 'dired-mode-hook 'gcr/dired-mode-hook)
```

After dabbling, something happend that really changed my mind. These three articles changed everything: [164] , [165] , [166]. They just made the power of Dired so obvious, and so easy to use, that it instantly because delightful to use. That was very, very cool. Even though I was really, really happy with Finder and Explorer... suddenly it just became so obvious and pleasant to use Dired. That is so wild.

Key notes when executing shell commands on file selection...

Substitution:

`<cmd> ?` 1* calls to cmd, each file a single argument

`<cmd> *` 1 call to cmd, selected list as argument

=<**cmd**> `*""`= have the shell expand the * as a globbign wildcard

- Not sure what this means

Synchronicity:

`<cmd> ...` by default commands are called synchronously

`<cmd> &` execute in parallel

`<cmd> ;` execute sequentially, asynchronously

`<cmd> ;&` execute in parallel, asynchronously

Key notes on working with files in multiple directories... use the following:

Use `find` just like you would at the command line and all of the results show up in a single Dired buffer that you may work with just like you would any other file appearing in a Dired buffer. The abstraction here becomes so obvious, you may ask yourself why you never considered such a thing *before* now (as I did):

```
(require 'find-dired)
(setq find-ls-option '("-print0 | xargs -0 ls -ld" . "-ld"))
```

Noting that:

`find-dired` is the general use case

`find-name-dired` is for simple, single string cases

And if you want to use the faster elisp version, that uses lisp regex, use:

`find-lisp-find-dired` for anything

`find-lisp-find-dired-subdirectories` for only directories

[164] `http://www.masteringemacs.org/articles/2014/04/10/dired-shell-commands-find-xargs-replacement/`
[165] `http://www.masteringemacs.org/articles/2011/03/25/working-multiple-files-dired/`
[166] `http://www.masteringemacs.org/articles/2013/10/10/wdired-editable-dired-buffers/`

Key notes on working with editable buffers...

As the author notes, you probably already instinctually knew what is possible. After reading his brief and concise exposition, it would be hard *not* to intuit what is possible! The options are big if you make a writable file buffer. Think about using multiple cursors. Done? Well, that is a no-brainer. Once you grok multiple cursors just `find-dired` what you need and then do what you need to do to it. Very cool.

`dired-toggle-read-only, C-x C-q` cycle between dired-mode and wdired-mode

`wdired-finish-edit, C-c C-c` commit your changes

`wdired-abort-changes, C-c ESC` revert your changes

```
(require 'wdired)
(setq wdired-allow-to-change-permissions t)
(setq wdired-allow-to-redirect-links t)
(setq wdired-use-interactive-rename +1)
(setq wdired-confirm-overwrite +1)
(setq wdired-use-dired-vertical-movement 'sometimes)
```

11. Save history of all things [167] , [168] , [169]

    It is nice to have commands and their history saved so that every time you get back to work, you can just re-run stuff as you need it. It isn't a radical feature, it is just part of a good user experience.

```
(let ((savehist-file-store "~/.emacs.d/savehist"))
  (when (not (file-exists-p savehist-file-store))
    (warn (concat "Can't seem to find a savehist store file where it was expected "
                  "at: " savehist-file-store " . Savehist should continue "
                  "to function normally; but your history may be lost.")))
  (setq savehist-file savehist-file-store))
(savehist-mode +1)
(setq savehist-save-minibuffer-history +1)
(setq savehist-additional-variables
      '(kill-ring
        search-ring
        regexp-search-ring))
```

12. Code folding [170] , [171]

    Code folding really isn't a hugely important function. You just use it once in a while and you notice it when you don't have it. For years I used this [172] and it is fine, but I figured I ought to stick with a more feature rich option, just to give it a try. Here are some of the other options: [173, 174, 175, 176]. If you know org-mode, then using that style of control makes it easier to use then the built in bindings for hideshow [177], on which hideshow-org is built.

[167] https://www.gnu.org/software/emacs/manual/html_node/emacs/Saving-Emacs-Sessions.html
[168] http://fly.srk.fer.hr/~hniksic/emacs/savehist.el
[169] https://stackoverflow.com/questions/1229142/how-can-i-save-my-mini-buffer-history-in-emacs
[170] http://www.emacswiki.org/emacs/HideShow
[171] http://gnufool.blogspot.com/2009/03/make-hideshow-behave-more-like-org-mode.html
[172] http://emacs.wordpress.com/2007/01/16/quick-and-dirty-code-folding/
[173] http://www.emacswiki.org/emacs/OutlineMode
[174] http://www.emacswiki.org/emacs/FoldingMode
[175] https://github.com/zenozeng/yafolding.el
[176] http://cedet.sourceforge.net/
[177] https://www.gnu.org/software/emacs/manual/html_node/emacs/Hideshow.html

```
(setq hs-hide-comments-when-hiding-all +1)
(setq hs-isearch-open +1)
(require 'hideshow-org)
(defun display-code-line-counts (ov)
  "Displaying overlay content in echo area or tooltip"
  (when (eq 'code (overlay-get ov 'hs))
    (overlay-put ov 'help-echo
                 (buffer-substring (overlay-start ov)
                                   (overlay-end ov)))))

(setq hs-set-up-overlay 'display-code-line-counts)
(defadvice goto-line (after expand-after-goto-line activate compile)
  "How do I get it to expand upon a goto-line? hideshow-expand affected block when using
  (save-excursion
    (hs-show-block)))
```

13. Copyright [178]

    Copyright management includes only two problems: keeping the near up to date and choosing the right one. The built in functions will insert a generic copyright and also update the year, and that is pretty nice. It would be nice to have something like this [179] created, though. Even an OSS license chooser would be nice [180], but I haven't found a nice option yet.

14. Spellchecking [181] , [182] , [183] , [184]

    There are two ways to spell-check: run-at-a-time or interactive. Both delegate the actual checking to aspell, ispell, and hunspell. Both styles are quite nice options, and flyspell will even integrated with compilers to help report those kinds of errors to you, too, but my personal preference for now is run-at-a-time. The taxpayers didn't pay so much to make flyspell have to do all the hard work for me. aspell is there most UNI*, running `ispell` from Emacs just does the right thing.

    Even after reading this later, I agree with it despite the fact that I constantly wax and wane between wanting to use it and finding something *better* despite having no criteria by which to truly judge in the first place.

```
(let ((aspell-dict "~/.aspell.en.pws"))
  (when (not (file-symlink-p aspell-dict))
    (warn
     (concat "aspell needs a symlink from " aspell-dict " to its true location. "
             "Please double check this. The fix might be as simple as: "
             "ln -s ~/git/bitbucket-grettke/home/.aspell.en.pws ~/.aspell.en.pws"))))
```

15. Binary file editing [185] , [186]

[178] https://www.gnu.org/software/emacs/manual/html_mono/autotype.html#Copyrights
[179] https://www.gnu.org/software/emacs/manual/html_node/elisp/Library-Headers.html
[180] http://choosealicense.com/
[181] https://www.gnu.org/software/emacs/manual/html_node/emacs/Spelling.html
[182] http://www.emacswiki.org/emacs/InteractiveSpell
[183] http://blog.binchen.org/posts/what-s-the-best-spell-check-set-up-in-emacs.html
[184] http://melpa.milkbox.net/#/ac-ispell
[185] https://www.gnu.org/software/emacs/manual/html_node/emacs/Editing-Binary-Files.html
[186] http://www.emacswiki.org/emacs/HexlMode

Long ago it was quite common to edit binary files if not for adding lots of cheats to games then to see CAFEBABE written in Java class files (if you can't enjoy either of those things then you are too serious). Hexl mode comes built into Emacs, and it is great to know that it is there.

16. Games [187]

    Sometimes you need a break, and you aren't a gamer any more, but that doesn't mean you can't have fun. `life` and `doctor` alone will give you something to ponder and practice not taking too seriously.

17. LaTeX [188] , [189]

    There is great support for LaTeX via AUCTeX and Ebib. For a while I used TeXWorks [190] and I was and remain very happy with it. Two things drew me back to Emacs for doing TeX primarily org-mode support for Ebib for managing my citation database. There is nothing more to it than that. This secion is a bit bare at the moment, as I will be filling it up as I move back to doing my work here. Perhaps a bigger project is figuring out where XeTeX fits in my future. [191] RefTeX looks too quite helpful [192] especially considering how well it seems to integrate with org-mode.

18. Command execution helper [193]

    When I call commands, I usually end up running the same commands over and over. There are of course keybdings to deal with this, and also command history. What I really prefer though is just being able to type an abbreviation for the command to access it, like `org-html-export-to-html` for example. Smex makes it happen.

    ```
    (require 'smex)
    (smex-initialize)
    ```

19. Location awareness

    The idea of reporting to you the current logical location within in the current file via information in the modeline [194] sounds very interesting to me. However, when I really thinkg about it, I have no good answer to the question: "If you didn't know how you got there, then how does it help to know that you are where you are?". That holds true at least, when it comes to maintaining files.

20. Rectangle / Cursors [195] , [196]

    Once in a very long while I have the need to modify rectangles. Only once in a while because one may use the key recorder to do most of the same work. There are a few options [197], and that bothers me, so I didn't choose any of them.

    Working here thought got me thinking about other folks perspectives, and I ended up here [198]. This is a strangely intriguing feature. It is quite versatile as long as you have got a mental model for

[187] https://www.gnu.org/software/emacs/manual/html_node/emacs/Amusements.html#Amusements
[188] https://www.gnu.org/software/auctex/
[189] http://ebib.sourceforge.net/
[190] https://www.tug.org/texworks/
[191] http://xetex.sourceforge.net/
[192] https://www.gnu.org/software/auctex/reftex.html
[193] https://github.com/nonsequitur/smex/
[194] https://www.gnu.org/software/emacs/manual/html_node/emacs/Which-Function.html
[195] https://www.gnu.org/software/emacs/manual/html_node/emacs/Rectangles.html
[196] https://github.com/magnars/multiple-cursors.el
[197] http://www.emacswiki.org/emacs/RectangleCommands
[198] http://emacsrocks.com/e13.html

things. The difference is that if you are OK with key macros, imagine that multiple-cursors is kind of a way to use keyboard macros while making it very visibile and dynamic and also using the cursor location along with that interactivity.

```
(require 'multiple-cursors)
```

21. Very large files

    Emacs will warn you if you open "large files" into a buffer. Thankfully, I have never had such an issue. However, surely I will have the need at some point in the future, and when I do I will look at vlfi [199]

22. Syntax checking

    It is a great feature. Flyspell never interested me though because of so many negative reports and it just didn't seem that important. Well, that was before breaking this document for the Nth time! There is a need, and Flycheck [200] seems to be the best of the best out there.

```
(require 'flycheck)
(add-hook 'after-init-hook #'global-flycheck-mode)
```

3.6.3 Application Modes [201] , [202]

When I set about on this project, I had ideas about how this document would look. The decent ideas worked out well. The good ideas were bad, and the unexpected ideas were delightful. I had though that athis section would be very graphical, but the more I work on it, the simpler it seems to be when you split it up in the respective sections. Originally I had wanted to use org tables for nearly everything, but now I question that desire (thought it is a great feature). The mistake that I made was not new to me and is suffered by all macro writers... the solution instead of waiting for experience and extracting it from there. As it turns out, I am human.

1. Monolith

   (a) Auto Modes [203]

```
(setq auto-mode-alist
      (append
       '(("\\.scm\\'" . scheme-mode)
         ("\\.rkt\\'" . scheme-mode)
         ("\\.ss\\'" . scheme-mode)
         ("\\.sls\\'" . scheme-mode)
         ("\\.sps\\'" . scheme-mode)
         ("\\.html\\'" . web-mode)
         ("\\.json\\'" . web-mode)
         ("\\.asc" . artist-mode)
         ("\\.art" . artist-mode)
         ("\\.asc" . artist-mode))
       auto-mode-alist))
```

[199] https://github.com/m00natic/vlfi
[200] https://github.com/flycheck/flycheck
[201] https://www.gnu.org/software/emacs/manual/html_node/emacs/Hideshow.html
[202] https://www.gnu.org/software/emacs/manual/html_node/emacs/Modes.html#Modes
[203] https://www.gnu.org/software/emacs/manual/html_node/elisp/Auto-Major-Mode.html

(b) All modes

Anything that should always happen goes here.

In the last setup, I went back and forth about where to do a and of line whitespace cleanup, if at all. The con is that with real-mode-autosave enabled, when you are typing your cursor keeps jumping, and that is not nice. An idle timer to do cleanup wouldn't be any different, because I really want saves to constantly be ocurring. My final decision is to just call `whitespace-cleanup` as needed rather than tracking down or writing some code to do it myself.

`fancy-narrow-to-region` [204] is a nice to have. Their approach is kind of worth noting, simply in that `;;;###autoload` commands are utilized so that the library is not required to be specified for use. Is this convenience without downside or just a bad idea from a support perspective? Unknown, at least for now. I'll err on the side of explicitness (when I'm paying attention at least).

```
(require 'fancy-narrow)
```

Make it real easy to utilize the things that imenu provides, but make it keyboard driven and available anywhere.

```
(require 'imenu-anywhere)
```

Auto-completion for .-separated words [205] seems like a good idea, so I will put it here and not worry too much about what header this lives in. The source explains how to use this feature... it must be specified what is allowed per-mode

- which makes sense.

```
(require 'auto-complete-chunk)
```

(c) Text

```
(defun gcr/text-mode-hook ()
  (rainbow-mode)
  (turn-on-real-auto-save)
  (fci-mode)
  (visual-line-mode)
  (gcr/untabify-buffer-hook))

(add-hook 'text-mode-hook 'gcr/text-mode-hook)
```

(d) Log edit / VC VC is the generalized version control suite for Emacs. It is quite nice and amazingly underappreciated. Elsewhere I make it easy to initiate a commit, and this makes it easier to finish it.

```
(defun gcr/log-edit-mode-hook ()
  "Personal mode bindings for log-edit-mode."
  (gcr/untabify-buffer-hook)
  (gcr/disable-tabs)
  (fci-mode +1))

(add-hook 'log-edit-mode-hook 'gcr/log-edit-mode-hook)
```

[204] https://github.com/Bruce-Connor/fancy-narrow

[205] https://github.com/tkf/auto-complete-chunk

```
(defun gcr/log-edit-mode-hook-local-bindings ()
  "Helpful bindings for log edit buffers."
  (local-set-key (kbd "C-;") 'log-edit-done))

(add-hook 'log-edit-mode-hook 'gcr/log-edit-mode-hook-local-bindings)
```

(e) Graphviz [206] , [207] , [208]

```
(defun gcr/graphviz-dot-mode-hook ()
  "Personal mode bindings for Graphviz mode."
  (fci-mode +1)
  (rainbow-mode)
  (visual-line-mode)
  (turn-on-real-auto-save))

(add-hook 'graphviz-dot-mode-hook 'gcr/graphviz-dot-mode-hook)

(let ((f (concat (cask-dependency-path gcr/cask-bundle 'graphviz-dot-mode)
                 "/graphviz-dot-mode.el")))
  (if (file-exists-p f)
      (load-file f)
    (warn "Could not locate a package file for Graphviz support. Expected it here (mi
```

(f) Lispy

```
(defconst lispy-modes '(emacs-lisp-mode-hook
                        ielm-mode-hook
                        lisp-interaction-mode-hook
                        scheme-mode-hook
                        geiser-repl-mode-hook))

(dolist (h lispy-modes)
  (add-hook h 'rainbow-mode))

(dolist (h lispy-modes)
  (when (not (member h '(ielm-mode-hook)))
    (add-hook h 'turn-on-smartparens-strict-mode)
    (add-hook h 'turn-on-pretty-mode)
    (add-hook h 'gcr/newline)
    (add-hook h 'turn-on-real-auto-save)
    (add-hook h 'gcr/untabify-buffer-hook)
    (add-hook h 'gcr/disable-tabs)
    (add-hook h 'fci-mode)
    (add-hook h 'hs-org/minor-mode +1)
    (add-hook h (function (lambda ()
                            (add-hook 'local-write-file-hooks
                                      'check-parens))))))
```

[206] http://www.graphviz.org/
[207] http://marmalade-repo.org/packages/graphviz-dot-mode
[208] http://orgmode.org/worg/org-contrib/babel/languages/ob-doc-dot.html

(g) Emacs Lisp

Make it obvious whether or not it is lexically scoped [209] or not and don't show that message whenever you enter a scratch buffer [210].

You may read more about Lisp Doc here [211].

You may read more about default console messages here [212].

```
(defun gcr/elisp-eval-buffer ()
  "Intelligently evaluate an Elisp buffer."
  (interactive)
  (gcr/save-all-file-buffers)
  (eval-buffer))

(defun gcr/elisp-mode-local-bindings ()
  "Helpful behavior for Elisp buffers."
  (local-set-key (kbd "s-l eb") 'gcr/elisp-eval-buffer)
  (local-set-key (kbd "s-l ep") 'eval-print-last-sexp)
  (local-set-key (kbd "s-l td") 'toggle-debug-on-error)
  (local-set-key (kbd "s-l mef") 'macroexpand)
  (local-set-key (kbd "s-l mea") 'macroexpand-all))

(require 'lexbind-mode)

(defun gcr/elisp-mode-hook ()
  (gcr/elisp-mode-local-bindings)
  (lexbind-mode)
  (turn-on-eldoc-mode))

(add-hook 'emacs-lisp-mode-hook 'gcr/elisp-mode-hook)

(setq initial-scratch-message nil)
```

(h) Scheme [213]

You should probably only use Geiser, forever. [214] Some day I would like to explore ac-geiser [215].

```
(require 'geiser)
(setq geiser-active-implementations '(racket))

(defun gcr/scheme-eval-buffer ()
  "Save and then evaluate the current Scheme buffer with Geiser."
  (interactive)
  (gcr/save-all-file-buffers)
  (geiser-mode-switch-to-repl-and-enter))

(defun gcr/scheme-mode-local-bindings ()
```

[209] http://marmalade-repo.org/packages/lexbind-mode
[210] https://www.gnu.org/software/emacs/manual/html_node/elisp/Startup-Summary.html
[211] https://www.gnu.org/software/emacs/manual/html_node/emacs/Lisp-Doc.html
[212] https://www.gnu.org/software/emacs/manual/html_node/elisp/Startup-Summary.html
[213] http://library.readscheme.org/index.html
[214] http://www.nongnu.org/geiser/
[215] https://github.com/xiaohanyu/ac-geiser

```
  "Helpful behavior for Scheme buffers."
  (local-set-key (kbd "<f5>") 'gcr/scheme-eval-buffer))

(add-hook 'scheme-mode-hook 'gcr/scheme-mode-local-bindings)
```

(i) Javascript [216] , [217]

```
(defun gcr/js-mode-hook ()
  (local-set-key (kbd "RET") 'newline-and-indent)
  (setq js-indent-level 2)
  (turn-on-real-auto-save)
  (fci-mode)
  (visual-line-mode)
  (gcr/untabify-buffer-hook))

(add-hook 'js-mode-hook 'gcr/js-mode-hook)

(let* ((ac-dir (cask-dependency-path gcr/cask-bundle 'auto-complete))
       (f (concat ac-dir "/dict/js-mode")))
  (when (not (file-exists-p f))
    (warn (concat
           "Could not locate a lib file for auto-complete JavaScript support. "
           "You might fix it with: ln -s " ac-dir "/dict/javascript-mode " f))))
```

(j) Web [218]

JSON support is included here, too. As you go about your business you read about features that seem nice or you really didn't actively think about. One such opportunity/mistake I am guilty of is using various web-based tools to accomplish thing for example formatting a JSON string, like this `{"foo":10, "bar":20, "baz":50}` example. For whatever reason, that is always just the way that I had done it. Reading irreal a nice option/reminder was posted for a JSON formatter [219]

```
(require 'web-mode)

(setq web-mode-enable-block-partial-invalidation t)

(setq web-mode-engines-alist
      '(("ctemplate" . "\\.html$")))

(defun gcr/web-mode-hook ()
  (whitespace-turn-off)
  (rainbow-turn-off)
  (visual-line-mode)
  (local-set-key (kbd "RET") 'newline-and-indent)
  (setq web-mode-markup-indent-offset 2)
  (setq web-mode-css-indent-offset 2)
  (setq web-mode-code-indent-offset 2)
```

[216] https://en.wikipedia.org/wiki/ECMAScript
[217] http://www.emacswiki.org/emacs/JavaScriptMode
[218] https://en.wikipedia.org/wiki/HTML
[219] https://github.com/gongo/json-reformat

```
  (setq web-mode-indent-style 2)
  (setq web-mode-style-padding 1)
  (setq web-mode-script-padding 1)
  (setq web-mode-block-padding 0)
  (gcr/untabify-buffer-hook))

(add-hook 'web-mode-hook 'gcr/web-mode-hook)

(require 'json-reformat)
```

(k) CSS

```
(defun gcr/css-modehook ()
  (fci-mode +1)
  (whitespace-turn-on)
  (rainbow-mode)
  (visual-line-mode)
  (gcr/untabify-buffer-hook)
  (turn-on-real-auto-save)
  (local-set-key (kbd "RET") 'newline-and-indent))

(add-hook 'css-mode-hook 'gcr/css-modehook)
```

(l) Make [220] , [221] , [222]

```
(defun gcr/make-modehook ()
  (fci-mode +1)
  (whitespace-turn-on)
  (rainbow-mode)
  (visual-line-mode)
  (turn-on-real-auto-save)
  (visual-line-mode)
  (local-set-key (kbd "RET") 'newline-and-indent))

(add-hook 'makefile-mode-hook 'gcr/make-modehook)
```

(m) Markdown [223]

Since org-mode exports to just about everything; my Markdown usage will be mostly limited to working with files on Github.

```
(autoload 'markdown-mode "markdown-mode"
  "Major mode for editing Markdown files" +1)

(add-to-list 'auto-mode-alist '("README\\.md\\'" . gfm-mode))
```

(n) Version control / Git [224] , [225]

[220] https://www.gnu.org/software/make/manual/make.html
[221] http://orgmode.org/worg/org-contrib/babel/languages/ob-doc-makefile.html
[222] http://www.emacswiki.org/emacs/MakefileMode
[223] http://jblevins.org/projects/markdown-mode/
[224] https://www.gnu.org/software/emacs/manual/html_node/emacs/Version-Control.html
[225] https://github.com/magit/magit

All version control systems basically work fine in Emacs version control (VC) abstraction layer, and I like it a lot.

What made me focus on Git and how I work with it though was two things: 1-I use that for hours and hours at work and home and 2-I had been using a standalone Git UI and I felt like it was kind of stupid not to use something built into Emacs. This will require furthe research.

One thing that I did find that I wanted though was that despite having set auto save to occur quite frequently, it was still possible to iniate a VC action without the buffering being saved. My solution for that is that before **every** VC action, at least the current buffer must be saved. This is OK because I believe that VC actions only occur on a per-file basis, versus command line VC operations. Then I added he same thing for diff.

```
(defadvice vc-next-action (before save-before-vc first activate)
  "Save all buffers before any VC next-action function calls."
  (gcr/save-all-file-buffers))

(defadvice vc-diff (before save-before-vc-diff first activate)
  "Save all buffers before vc-diff calls."
  (gcr/save-all-file-buffers))

(defadvice vc-revert (before save-before-vc-revert first activate)
  "Save all buffers before vc-revert calls."
  (gcr/save-all-file-buffers))
```

Something that I never missed from Idea was version control status info in the fringe, just never used it. Then when I saw it [226] in Emacs, I got curious about how it *may* be used. So, I installed it. Curious to see how it will facilitate communicating the status of this document. Initial experiences has me thinking that it is actually much nicer than I figured, so I will enable it globally for a while.

```
(diff-hl-mode)
```

(o) LilyPond [227]

All of my experience with musical notation is through GuitarPro [228] and even there I'm a baby user. Despite that, I've been curious about music theory for a long time and this seems like a good way to take a dip. At first, it didn't seem to work. then I "rebooted" and it seemed to work, but not in org HTML export. This will require furthe research.

(p) Line Wrapping [229] , [230] / Line breaking [231]

A long time ago I disabled line-wrapping because I kept all of my files less than 80 lines and life was simple. This approach actually worked fine for a long, long time, that was until it quit working well, when I started working on Emacs a lot of the time on different machines with different resolutions. Perhaps there was more too it then this, but I really don't recall. Reading up on it now, it seems that there is a nice option built in and I didn't have much to consider other than to capture my decisions on how I want the wrapping to work. Basically I don't want indicators since I have line numbers it is obvious, and I will just make the mode global and

[226] https://github.com/dgutov/diff-hl

[227] http://lilypond.org/

[228] http://www.guitar-pro.com/en/index.php?pg=accueil-2&utm_expid=13369301-5.jyDTwdKfQ_CCdEqtpCIynQ.1&utm_referrer=https%3A%2F%2Fwww.google.com%2F

[229] https://www.gnu.org/software/emacs/manual/html_node/emacs/Visual-Line-Mode.html

[230] http://www.emacswiki.org/emacs/VisualLineMode

[231] https://www.gnu.org/software/emacs/manual/html_node/emacs/Auto-Fill.html

disable it in cases where I need to do so. Reading up on it more I figured that enabling it for text modes was the simplest thing, and I'll tweak it from there.

```
(global-visual-line-mode 1)
(diminish 'visual-line-mode)
(diminish 'global-visual-line-mode)
```

Another option available to us is to *simply* break the line once it reaches a pre-set length using Auto fill mode. One might break all lines at 80 characters for example. Although that is a nice option, I prefer dealing with it manually so I know what is happening in any particular buffer, and visual line mode makes that loads easier.

(q) Emacs Speaks Statistics (ESS) [232] , [233] , [234] , [235] , [236] , [237]

For a minimalist release history, read the news file [238].

For a brief, brief overview and release history, read the readme [239].

For a comprehensive overview, read the manual [240]. In it:

- `S` refers to any language in the family. `R` is what I'm interested in.
- First 2.5 pages do some nice expectation-setting.
- Generally seems like a highly rich development environment with support for

editing, debugging, and supporting with everything that you would expect from the best of Emacs.

- Manual covers most request variables for configuring, but the Customize

facility covers more, and mentions that either way you should avoid doing so until you have used ESS for a while.

- Check that ESS is installed with a call to `ess-version`.

R notes, or notices:

- `R` will start R inside Emacs
- Multiple ESS processes may run simultaneously, and may be selected by a

specific language via their buffer name that has a number appended, or may be accessed via a menu using `ess-request-a-process`.

- ESS works transparently on remote machines using TRAMP for manage a remote

R instance. An example is provided for Amazon. Means exist for supporting remote graphicals displays or redirecting to a file. Excellent support seems to exist to quite flexibly support unexpected things like stating an ESS supported program in a plain old shell and being able to convert it to an ESS supported buffer.

Get ESS loaded before randomy doing other stuff:

```
(require 'ess-site)
```

[232] `http://ess.r-project.org/`
[233] `https://en.wikipedia.org/wiki/Emacs_Speaks_Statistics`
[234] `http://www.emacswiki.org/emacs/EmacsSpeaksStatistics`
[235] `http://blog.revolutionanalytics.com/2011/08/ess.html`
[236] `http://blog.revolutionanalytics.com/2014/03/emacs-ess-and-r-for-zombies.html`
[237] `https://rstudio-pubs-static.s3.amazonaws.com/2246_6f220d4de90c4cfda4109e62455bc70f.html`
[238] `http://ess.r-project.org/Manual/news.html`
[239] `http://ess.r-project.org/Manual/readme.html`
[240] `http://ess.r-project.org/Manual/ess.html`

Interaction stuff:

- Return sends the input from wherever you hit return, nice.
- `M-{` and `M-}` cycle through commands you ran
- `M-h` select a whole "paragraph", a block in their terms
- `C-x [` moves through the previous ESS sessions, `C-x ]` forward.
- `C-c C-p` and `C-c C-n` cycle through previous commands.
- `C-c RET` copies an old command to the prompt without running it
- Keep your session transcript pruned
- `ess-transcript-clean-region` removes non-commands from a transcript for you
- Previous command lookup can be done by completion via `comint-*-matching`.

`M-p` and `M-n` seem to work just fine though.

- Previous command execution, by name, offset, or just the last one, are by `!`

This feature is actually quite rich and a real regexen style system.

- Always show eldoc for R stuff, everywhere it may.

```
(setq ess-eldoc-show-on-symbol t)
```

Interaction stuff

- Show objects in the workspace: `C-c C-x`
- Search for what libs are available to the workspace: `C-c C-s`
- Load file with source: `C-c C-l`
- Visit errors: =C-c '= and =C-x '=
- Show help on an object: `C-c C-v`
- Quit: `C-c C-q`
- Abort: `C-c C-c`
- Switch between the console and the most recent file buffer: `C-c C-z`

Sending code to the ESS process

- `ess-eval-region-or-line-and-step`: Eval the region, or the line, move to next line
- `C-M-x`: Eval the current region, function, or paragraph.
- `C-c C-c`: Do that and then go to the next line.
- `C-c C-j`: Eval the current line
- `C-c M-j`: Eval line and jump to the console
- `C-c C-f`: Eval the currently selected function
- `C-c M-f`: Eval the currently selection function and jump to the console
- `C-c C-r`: Eval the region
- `C-c M-r`: Eval the region and jump to the console
- `C-c C-b`: Eval the buffer
- `C-c M-b`: Eval the buffer and jump to the console
- You can do all this stuff from transcript files, too. My thought is that I never, ever will and if I do need to, I'm looking up the commands again as I don't want to make a habit of doing that kind of thing (running old transcripts).

Editing objects and functions:

- `C-c C-e C-d`: Edit the current object
- `C-c C-l`: Load source file into the ESS process
- `TAB` Indents/reformats or completes code.
- `M-;`: Correctly indents the current comment

Getting help:

- `C-c C-v`: `ess-display-help-on-object`: Get help on anything
- `?`: Show commands available in help mode
- `h`: Show help for a different object. Currently focused object defaults.
- `n` and `p`: Cycle through sectons
- `l`: Eval the current line in the console; usually sample code.
- `r`: Eval current region, too
- `q`: Quit out of that buffer
- `k`: Kill that buffer
- `x`: Kill that buffer and return to ESS
- `i`: Get info on a package
- `v`: Show vignettes
- `w`: Show current help page in borwser

Completion:

- `TAB`: Complete anything
- `M-?`: Show completions available
- `ess-resynch`: Refreshes the completion cache

Debugging:

- Full featured debugger
- `M-C`: Continue
- `M-N`: Next step
- `M-U`: Up frame
- `M-Q`: Quit

```
(setq ess-use-tracebug t)
```

Breakpoints:

- `b`: BP (repeat to cycle BP type)
- `B`: Set conditional BP
- `k`: Kill BP
- `K`: Kill all BPs
- `o`: Toggle BP state
- `l`: Set logger BP
- `n`: Goto next BP
- `p`: Goto previous BP

Debugging; be sure to read this [241]:

- `‘`: Show traceback

[241] `https://code.google.com/p/ess-tracebug/`

- `~`: Show callstack
- `e`: Toggle error action (repeat to cycle)
- `d`: Flag for debugging
- `u`: Unflag for debugging
- `w`: Watch window
- Be sure to specify this per-project.

```
(setq ess-tracebug-search-path '())
```

- Make error navigation simpler

```
(define-key compilation-minor-mode-map [(?n)] 'next-error-no-select)
(define-key compilation-minor-mode-map [(?p)] 'previous-error-no-select)
```

- The font size for watched variables.

```
(setq ess-watch-scale-amount -1)
```

- Make it easier to know what object values are.

```
(setq ess-describe-at-point-method 'tooltip)
```

- Rdired is another way to work with object

```
(autoload 'ess-rdired "ess-rdired")
```

- Visualize data frames better.

```
(require 'ess-R-data-view)
```

- Visualize just about anything.

```
(require 'ess-R-object-popup)
(define-key ess-mode-map "\C-c\C-g" 'ess-R-object-popup)
```

Documentation

- Whole section on native documentation; I'll re-visit as needed.
- Roxygen, too.

`ess-developer` helps you to easily work within specific namespaces.
Rutils: keybindings to aid real usage

- `C-c C-.` `l`: List all packages in all available libraries.
- `C-c C-.` `r`: List available packages from repositories listed by getOptions("repos")

in the current R session.

- `C-c C-.` `u`: Update packages in a particular library lib and repository repos.
- `C-c C-.` `a`: Search for a string using apropos.
- `C-c C-.` `m`: Remove all R objects.
- `C-c C-.` `o`: Manipulate R objects; wrapper for ess-rdired.
- `C-c C-.` `w`: Load a workspace file into R.
- `C-c C-.` `s`: Save a workspace file.
- `C-c C-.` `d`: Change the working directory for the current R session.
- `C-c C-.` `H`: Use browse-url to navigate R html documentation.

`ess-mode-silently-save` is worth a million bucks; usually I have to hand code this.

As of *<2014-01-31 Fri>*, you need to manually load ESS when you pull it from MELPA [242]. That is totally fine with me, that is really the best way to load stuff. Out of curiousity, I read more about it here [243], but that occurred before this previous post made by the maintainers. Even the source code in `ess-autoloads.el` has a license from 2012, which is before the aforementioned post. As such, this configuration step seems correct and necessary for now. Additionally, this how the user manual expects a typical manual setup to be configured.

Looked a tiny bit at how R hackers are formatting their code [244] , [245]. The simple (dumb) part of me suspects that C++ formatting is generally just fine [246].

There is strangely nice discussion about where temp files may be stored; specifically for cases where you edit samely-named objects and want to keep them in the same directory but per-project. That is not the need now, and it is nice to know that it is an option.

```
(setq gcr/r-dir "~/.R/")

(setq inferior-ess-program "R")
(setq inferior-R-program-name "R")
(setq ess-local-process-name "R")
(setq inferior-ess-own-frame nil)
(setq inferior-ess-same-window t)
(setq ess-ask-for-ess-directory nil)

(setq comint-scroll-to-bottom-on-input 'this)
(setq comint-scroll-to-bottom-on-output 'others)
(setq comint-show-maximum-output t)
(setq comint-scroll-show-maximum-output t)
(setq comint-move-point-for-output t)
(setq comint-prompt-read-only t)

(setq ess-history-directory gcr/r-dir)

(setq ess-execute-in-process-buffer +1)
(setq ess-switch-to-end-of-proc-buffer t)
(setq ess-eval-visibly nil)

(setq ess-tab-complete-in-script +1)
(setq ess-first-tab-never-complete 'symbol-or-paren-or-punct)

(setq ess-source-directory gcr/r-dir)

(setq ess-help-own-frame nil)

(setq ess-use-ido t)

(add-to-list 'auto-mode-alist '("\\.rd\\'" . Rd-mode))
```

[242] https://stat.ethz.ch/pipermail/ess-help/2014-January/009705.html
[243] https://github.com/milkypostman/melpa/issues/6
[244] https://google-styleguide.googlecode.com/svn/trunk/Rguide.xml
[245] http://adv-r.had.co.nz/Style.html
[246] https://stackoverflow.com/questions/7502540/make-emacs-ess-follow-r-style-guide

```
(setq ess-use-eldoc t)
(setq ess-eldoc-show-on-symbol t)
(setq ess-eldoc-abbreviation-style 'normal)

(local-set-key (kbd "C-c C-. S") 'ess-rutils-rsitesearch)

(require 'ess-rutils)
(setq ess-rutils-keys +1)

(require 'r-autoyas)

(setq r-autoyas-debug t)
(setq r-autoyas-expand-package-functions-only nil)
(setq r-autoyas-remove-explicit-assignments nil)
(setq r-autoyas-number-of-commas-before-return 0)

(defun gcr/ess-mode-hook ()
  (ess-set-style 'RRR 'quiet)
  (turn-on-pretty-mode)
  (local-set-key (kbd "<f7>") 'ess-describe-object-at-point)
  (r-autoyas-ess-activate)
  (visual-line-mode)
  (lambda () (add-hook 'ess-presend-filter-functions
                       (lambda ()
                         (display-warning
                          'ess-mode
                          "ESS now supports a standard pre-send filter hook. Please u
                          :warning)))))

(add-hook 'ess-mode-hook 'gcr/ess-mode-hook)

(defun gcr/Rd-mode-hook ()
  (gcr/ess-mode-hook))

(add-hook 'Rd-mode-hook 'gcr/Rd-mode-hook)

(add-to-list 'auto-mode-alist '("\\.Rmd$" . r-mode))

(defun gcr/inferior-ess-mode-hook ()
  (gcr/ess-mode-hook))

(add-hook 'inferior-ess-mode-hook 'gcr/inferior-ess-mode-hook)
```

- This [247] post shares a nice setup for the assignment key; primarily if

you use underscores in your variable names, which I do on occassion. After coding like this for just 10 short minutes it drove me nuts and that is totally counter intuitive to me; I never would have expected that having to type two characters to do an assignment would give me

[247] http://www.r-bloggers.com/a-small-customization-of-ess/

nuts. Anyway, the default behavior is just fine; hit underscore twice gives you an underscore, and one gives you an assignment!

Philosphy

The current ESS maintainers philosophies about how to maintain an R codebase make sense to me and are virually the same as my own. Quite simply, the rule is that the code artifacts are the single source of system definition. Consequently, the system should be configured in this manner:

- We want to keep dump files after loading them; never delete them.

```
(setq ess-keep-dump-files +1)
```

- ESS allows us to quite easily modify live S objects and functions. It provides

this functionality via `ess-dump-object-into-edit-buffer`. Thee changes are considered to be experimental, and not part of the master record according to our philosophy. As such, we don't care to know that these new version ever existed and their record will be forgotten from history. In other words, that new, modified version of the object or function, is never saved to a file for later reuse.

```
(setq ess-delete-dump-files nil)
```

- Since our systems are entirely file-based, the entirey of the system most

likely lives in different files. Before loading any file for sourcing, save any ESS source buffers. This approach is in addition to two other things: (1) Emacs is auto-saving every file buffer quite frequently and (2) there is advice before every manual `eval` call so that the buffers and their files stay in sync. Yes, it is really that important.

```
(setq ess-mode-silently-save +1)
```

- During the experimental mode of system development, you are likely to hack on

things using an ESS buffer associated with a file. Things can happen quite unexpectedly, and it is easier to know that the code that you have `eval`'d is the value that is actually currently saved on-disk. You get it by now, that is my personal preference. It is just a lot easier IMHO to know that your files are persisted and my be stored in your VCS and that things "look are right".

```
(defadvice ess-eval-region-or-line-and-step (before before-ess-eval-region-or-line-an
  (gcr/save-all-file-buffers))

(defadvice ess-eval-region-or-function-or-paragraph (before before-ess-eval-region-or
  (gcr/save-all-file-buffers))

(defadvice ess-eval-region-or-function-or-paragraph-and-step (before before-ess-eval-
  (gcr/save-all-file-buffers))

(defadvice ess-eval-line (before before-ess-eval-line activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-line-and-go (before before-ess-eval-line-and-go activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-function (before before-ess-eval-function activate)
```

```
  (gcr/save-all-file-buffers))

(defadvice ess-eval-function-and-go (before before-ess-eval-function-and-go activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-region (before before-ess-eval-region activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-region-and-go (before before-ess-eval-region-and-go activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-buffer (before before-ess-eval-buffer activate)
  (gcr/save-all-file-buffers))

(defadvice ess-eval-buffer-and-go (before before-ess-eval-buffer-and-go activate)
  (gcr/save-all-file-buffers))
```

- Don't save the workspace when you quit R and don't restore **ANYTHING** when you start it, either.

```
(setq inferior-R-args "--no-save --no-restore")
```

- Indent curly brackets correctly.

Smartparens is serving me well. In this mode it is for curly, round, and square brackets. ESS handles indenting mostly right, too. One thing was unpleasant, though. When you define a new function, hitting return brings the curely bracket down to the newline but doesn't give it an empty line and indent the cursor one indentation over so that you may begin implementing the function. That is a big hassle; 4 unnecessary keystroke, it is really distracting and takes you out of the flow. It is such a little thing yet it is so powerfully distracting. It is like a mosquito in your tent! Searching for a solution revealed that I am not alone here.

This post [248] handles brackets, but not indentation. ESS itself handles indentation quite well [249] but doesn't provide the behavior that I want. This post [250] captured exactly what I was facing, yet didn't end with a solution which was kind of shocking. Searching some more I ended up here [251], and this seems like the ideal solution by the author of smartparens himself. This is probably a common thing as I found another post with exactly my situation referencing that aforementioned solution, too [252]. This is a nice generalizable approach that should serve me well for just about everything in this solution-area. Here [253] is a post showing a more advanced usage that handles context which is nice to know is an option.

```
(sp-local-pair 'ess-mode "{" nil :post-handlers '((gcr/indent-curly-block "RET")))
```

(r) Info [254]

[248] http://www.emacswiki.org/emacs/ESSAutoParens

[249] http://emacs.1067599.n5.nabble.com/indentation-not-working-if-parentheses-are-already-closed-td283806.html

[250] https://stackoverflow.com/questions/18420933/enabling-mode-specific-paren-indentation-in-emacs-prelude

[251] https://github.com/Fuco1/smartparens/issues/80

[252] https://github.com/bbatsov/prelude/issues/374

[253]: https://github.com/rdallasgray/graphene/blob/master/graphene-smartparens-config.el

[254] http://www.emacswiki.org/emacs/InfoMode

Once you accept Emacs and learn to enjoy Info files you may want to be able to navigate them quickly, even if you haven't read the user manual as I have not. ace-link [255] is really a nice way to do that.

```
(ace-link-setup-default)
```

(s) Vagrant [256]

Vagrant is quite nice. Perhaps a bit preemptively, I'm trying to get Emacs setup nice for what I already know I must do.

This belongs in this heading I believe:

```
(add-to-list 'auto-mode-alist '("Vagrantfile$" . ruby-mode))
```

Nice package [257] for working with Vagrant; hundreds of people already using it. No configuration even necessary.

This package [258] is also quite nice:

```
(eval-after-load 'tramp
  '(vagrant-tramp-enable))
```

(t) Ruby [259]

My first setup of Ruby is primarily for Vagrant, so I didn't dig super deep into the options. The defaults will be just fine. The stuff that I commonly use may eventually want to end up in `prog-mode`, but I'm still not sure what really uses that and how I may being refactoring.

```
(defun gcr/ruby-mode-hook ()
  (fci-mode +1)
  (rainbow-mode)
  (gcr/untabify-buffer-hook)
  (turn-on-real-auto-save)
  (visual-line-mode)
  (local-set-key (kbd "RET") 'newline-and-indent))

(add-hook 'ruby-mode-hook 'gcr/ruby-mode-hook)
```

(u) Eshell [260] , [261] , [262]

If you've never learned bash or korn or c-shell, then you are missing out on having some good fun... I mean work, getting work done. That said, I'm a baby when it comes to really using them. It seemed like a good idea to learn some of them well, and one that works seamnlessly with Emacs seems like a great idea. Since it is just another Elisp program, it has access to the same scope as everything else running inside Emacs. The resources on this tool are a bit varied and all valuable so I included all of them. The big takeaway is that you've got a "normal" looking shell interface whose commands work transparently with Elisp commands... and that can bey very pleasant.

Command completion is available. Commands input in eshell are delegated in order to an alias, a built in command, an Elisp function with the same name, and finally to a system call.

[255] https://github.com/abo-abo/ace-link
[256] http://www.vagrantup.com/
[257] https://github.com/ottbot/vagrant.el
[258] https://github.com/dougm/vagrant-tramp
[259] https://www.ruby-lang.org/en/
[260] https://www.gnu.org/software/emacs/manual/html_mono/eshell.html
[261] http://www.masteringemacs.org/articles/2010/12/13/complete-guide-mastering-eshell/
[262] http://www.khngai.com/emacs/eshell.php

Semicolons deparate commands. `which` tells you what implementation will satisfy the call that you are going to make. The flag `eshell-prefer-lisp-functions` does what it says. `$$` is the result of the last command. Aliases live in `eshell-aliases-file`. History is maintained and expandable. `eshell-source-file` will run scripts. Since Eshell is not a terminal emulator, you need to tell it about any commands that need to run using a terminal emulator, like anything using curses by adding it to to `eshell-visual-commands`.

i. Control Files

```
alias clear recenter 0
alias d 'dired $1'
alias g git $*
alias gb git branch $*
alias gco git checkout $*
alias gpom git push origin master
alias gst git status
alias la ls -lha $*
alias ll ls -lh $*
alias s ssh $*
alias top proced
```

ii. Config [263] , [264]

```
(setq eshell-prefer-lisp-functions nil
      eshell-cmpl-cycle-completions nil
      eshell-save-history-on-exit t
      eshell-cmpl-dir-ignore "\\`\\(\\.\\.?\\|CVS\\|\\.svn\\|\\.git\\)/\\'")

(eval-after-load 'esh-opt
  '(progn
     (require 'em-cmpl)
     (require 'em-prompt)
     (require 'em-term)
     (setenv "PAGER" "cat")
     (add-hook 'eshell-mode-hook
               (lambda ()
                 (message "Protovision... I have you now.")
                 (setq pcomplete-cycle-completions nil)))
     (add-to-list 'eshell-visual-commands "ssh")
     (add-to-list 'eshell-visual-commands "tail")
     (add-to-list 'eshell-command-completions-alist
                  '("tar" "\\(\\.tar|\\.tgz\\|\\.tar\\.gz\\)\\'"))))

(let ((eshell-dir "~/.emacs.d/eshell"))
  (when (not (file-symlink-p eshell-dir))
    (warn
     (concat "eshell needs a symlink from " eshell-dir " to its true location. "
             "Please double check this. The fix might be as simple as: "
             "ln -s ~/git/bitbucket-grettke/home/eshell/ ~/.emacs.d/eshell"))))

(setq eshell-prompt-regexp "^.+@.+:.+> ")
```

[263] http://eschulte.github.io/emacs-starter-kit/starter-kit-eshell.html

[264] https://github.com/bbatsov/emacs-dev-kit/blob/master/eshell-config.el

```
(setq eshell-prompt-function
      (lambda ()
        (concat
         (user-login-name)
         "@"
         (system-name)
         ":"
         (eshell/pwd)
         "> ")))
```

(v) IRC [265] , [266] , [267] , [268] , [269]

For a while I used MIRC [270], and then Irssi [271] and both are very nice. Since I hang around in Emacs all day though, I figured I ought to take a look at ERC. The documentation is highly modular, reflecting how the application itself is implemented. Patience may be required, but the reward for it is brilliant.

The main configuration is quite straightforward, with many ways to do it.

```
(require 'erc)

(setq gcr/erc-after-connect-hook-BODY nil)

(defun gcr/erc-after-connect-hook ()
  (gcr/erc-after-connect-hook-BODY))

(add-hook 'erc-after-connect 'gcr/erc-after-connect-hook)

(defun gcr/irc ()
  "Connect to my preferred IRC network."
  (interactive)
  (let ((file "~/.irc.el"))
    (when (not (file-exists-p file))
      (warn (concat "Can't seem to find an ERC credential file at: " file)))
    (with-temp-buffer
      (insert-file-contents file)
      (let ((grettke-irc-freenode-net-password (buffer-string)))
        (erc
         :server "irc.freenode.net"
         :port "6667"
         :nick "grettke"
         :password grettke-irc-freenode-net-password
         :full-name "Grant Rettke")
        (let ((gcr/erc-after-connect-hook-IMPL
               (lambda ()
                 (message "It ran...")
```

[265] http://mwolson.org/static/doc/erc.html
[266] http://emacs-fu.blogspot.com/2009/06/erc-emacs-irc-client.html
[267] http://edward.oconnor.cx/config/.ercrc.el
[268] http://www.shakthimaan.com/posts/2011/08/13/gnu-emacs-erc/news.html
[269] https://gitcafe.com/Darksair/dotfiles-mac/blob/master/.emacs-erc.el
[270] http://www.mirc.com/
[271] http://www.irssi.org/

```
                (erc-message
                 "PRIVMSG"
                 (concat "NickServ identify "
                         grettke-irc-freenode-net-password)))))
          (setq gcr/erc-after-connect-hook-BODY gcr/erc-after-connect-hook-IMPL))))))
```

```
(define-key erc-mode-map (kbd "C-c C-RET") 'erc-send-current-line)
```

The remaining configuration areas for modules that provide additional ERC functionality. Some of them are automatically loaded for you, some are not.

Autoaway [272] automatically marks you away or present after a desired timespan.

```
(require 'erc-autoaway)
(add-to-list 'erc-modules 'autoaway)
(setq erc-autoaway-idle-seconds 600)
(setq erc-autoaway-message "autoaway just demanded that I step out now")
(setq erc-auto-set-away +1)
(erc-update-modules)
```

Autojoin [273] automatically joins you to your preferred channels

```
(require 'erc-join)
(erc-autojoin-mode +1)
(setq erc-autojoin-channels-alist
      '((".*freenode.net" "#emacs" "#org-mode" "#scheme" "#r")))
```

Button [274] gives you clickable button-based events for various types of objects.

```
(require 'erc-button)
(erc-button-mode +1)
(setq erc-button-wrap-long-urls  nil
      erc-button-buttonize-nicks nil)
```

Completion [275] for various fields is provided by pcomplete and requires no configuration.

Wrap [276] long lines in the buffer.

```
(require 'erc-fill)
(erc-fill-mode +1)
(setq erc-fill-column 72)
(setq erc-fill-function 'erc-fill-static)
(setq erc-fill-static-center 0)
```

IRC control characters [277] may be made visible.

```
(erc-irccontrols-enable)
```

List [278] lists channels nicely and requires no configuration.

Match [279] highlights things that you care about.

[272] http://www.emacswiki.org/emacs/ErcAutoAway
[273] http://www.emacswiki.org/emacs/ErcAutoJoin
[274] http://www.emacswiki.org/emacs/ErcButton
[275] http://www.emacswiki.org/emacs/ErcCompletion
[276] http://www.emacswiki.org/emacs/ErcFilling
[277] http://www.opensource.apple.com/source/emacs/emacs-84/emacs/lisp/erc/erc.el
[278] https://github.com/pymander/erc/blob/master/erc-list.el
[279] http://www.emacswiki.org/emacs/ErcMatch

```
(setq erc-current-nick-highlight-type 'keyword)
(setq erc-pals '("leppie"))
(setq erc-fools '("lamer" "dude"))
(remove-hook 'erc-text-matched-hook 'erc-hide-fools)
```

Netsplits [280] occur when an IRC server is disconnected.

```
(require 'erc-netsplit)
(erc-netsplit-mode 1)
```

Noncommands lets you ignore command output of non-IRC related commands. For now I don't uses any, but wanted to note this feature.
Notify [281] notifies you when you are messaged.

```
(add-to-list 'erc-modules 'notify)
(erc-update-modules)
```

Handle paging [282] from other users.

```
(add-to-list 'erc-modules 'page)
(require 'erc-page)
(erc-page-mode 1)
(erc-update-modules)
```

Ring [283] gives you a command history.

```
(require 'erc-ring)
(erc-ring-mode 1)
```

Scrolltobottom [284] keeps your prompt line at the bottom.

```
(add-to-list 'erc-modules 'scrolltobottom)
(erc-update-modules)
```

Timestamp [285] nicely shows you when messages occurred.

```
(add-to-list 'erc-modules 'stamp)
(require 'erc-stamp)
(erc-stamp-mode 1)
(setq erc-insert-timestamp-function      'erc-insert-timestamp-left
      erc-timestamp-only-if-changed-flag t
      erc-timestamp-format               "[%H:%M] "
      erc-insert-away-timestamp-function 'erc-insert-timestamp-left
      erc-away-timestamp-format          "<%H:%M> ")
(erc-update-modules)
```

Tracking [286] of channels helps you know what is happening on closed windows.

[280] http://www.emacswiki.org/emacs/ErcNetsplit
[281] http://www.emacswiki.org/emacs/ErcNickNotify
[282] https://github.com/emacsmirror/erc/blob/master/erc-page.el
[283] https://github.com/emacsmirror/erc/blob/master/erc-ring.el
[284] http://www.emacswiki.org/emacs/ErcScrollToBottom
[285] http://www.emacswiki.org/emacs/ErcStamp
[286] http://www.emacswiki.org/emacs/ErcChannelTracking

```
(add-to-list 'erc-modules 'track)
(require 'erc-track)
(setq erc-track-switch-direction 'importance)
(setq erc-track-exclude-types
      '("324" "329" "332" "333" "353"
        "JOIN" "NAMES" "NICK" "QUIT" "PART" "TOPIC"))
(setq erc-track-position-in-mode-line +1)
(defvar erc-channels-to-visit nil
  "Channels that have not yet been visited by erc-next-channel-buffer")
(defun erc-next-channel-buffer ()
  "Switch to the next unvisited channel. See erc-channels-to-visit"
  (interactive)
  (when (null erc-channels-to-visit)
    (setq erc-channels-to-visit
          (remove (current-buffer) (erc-channel-list nil))))
  (let ((target (pop erc-channels-to-visit)))
    (if target
        (switch-to-buffer target))))
(erc-update-modules)
```

Tweet [287]: Show inlined info about youtube links in erc buffers.

```
(require 'erc-tweet)
(add-to-list 'erc-modules 'tweet)
(erc-update-modules)
```

Image [288]: Show inlined images (png/jpg/gif/svg) in erc buffers.

```
(require 'erc-image)
(add-to-list 'erc-modules 'image)
(erc-update-modules)
```

Youtube [289]: Show inlined info about youtube links in erc buffers.

```
(require 'erc-youtube)
(add-to-list 'erc-modules 'youtube)
(erc-update-modules)
```

Highlight nicks [290]: ERC Module to Highlight Nicknames.

```
(require 'erc-hl-nicks)
(add-to-list 'erc-modules 'hl-nicks)
(erc-update-modules)
```

i. Libraries

A. Generally nice

- Dash / Dash-Functional

[287] https://github.com/kidd/erc-tweet.el
[288] https://github.com/kidd/erc-image.el
[289] https://github.com/kidd/erc-youtube.el
[290] https://github.com/leathekd/erc-hl-nicks

Not totally sure where this belongs, but dash [291] is something that a lot of the packages I use require, and it is an excellent library, so it needs recognition in this document, and the Cask file, too.

```
(eval-after-load "dash" '(dash-enable-font-lock))
```

- f [292]
- s [293]

B. Built-in
pcase [294] provides pattern-matching macros. This is very nice whether you've already used something like this before, or not!

C. Characters / Unicode
unidecode [295] does its best to convert UTF-8 to ASCII; then I found that it wouldn't load so I removed it.
If you've verused a character terminal then you already know that figlet [296] is a mandatory tool. This [297] package makes it nice to use. It has all the stuff you *would* want to do, like figletify stuff. It even has a little helper function to show you how the fonts look... beccause you *know* that you would have ended up writing something like that yourself if it wreren't here:

```
(require 'figlet)
```

(w) Apropos mode
Why did I wait to long to learn how pleaasant this is to use?! There is of course a mode for it, with a hook/

2. Module

(a) Diagramming, UML creation, Workflow
How you perform these taks is entirely up to you. There are a lot of good options both inside and outside of Emacs. For the general cases, I like the ones that are built in and play nice, especially with org-mode. At its simplest, artist-mode [298] is plenty fine for diagramming and stuff. Graphviz also works well [299]. Ditaa is sort of the next level up [300], and finally PlantUML [301]. They are all good options at different times, and they all work with org-mode. Everything I will publish will go through org-mode. org-modes just shines so, so brightly.
As of writing, I'm undecided onw how best to standarding on a solution in this area. The good thing is that each tool is a good fit depending upon what you want to accomplish:

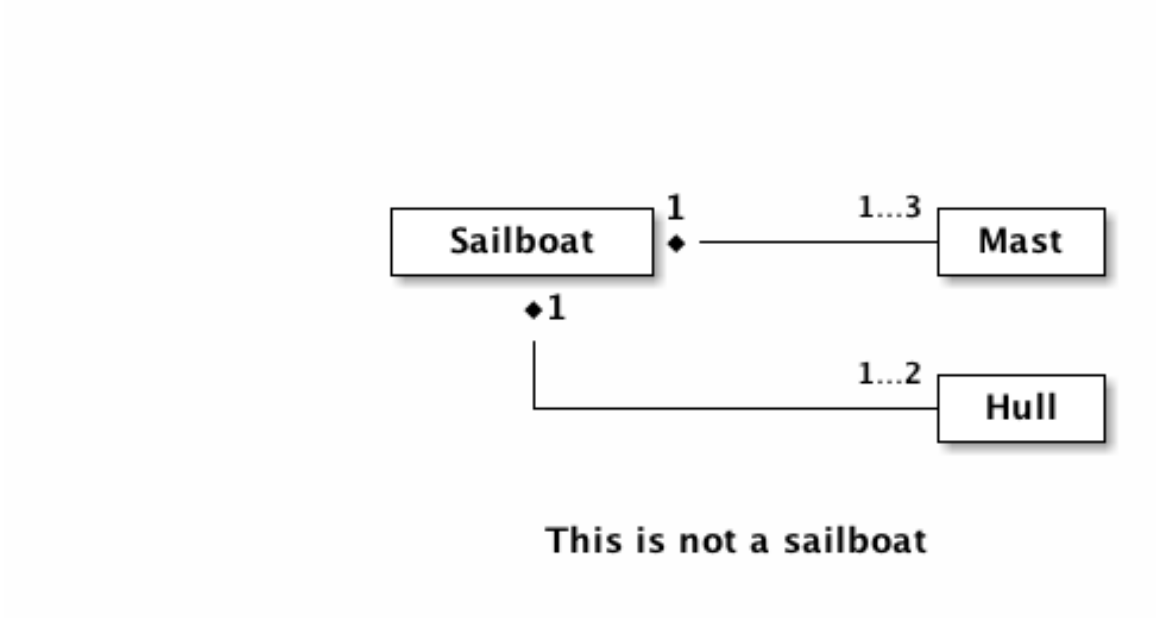
- artist-mode: Anything in ultra portable text, asii or utf-8, just works.
- Graphviz: Graphicaly and lays things out automatically.
- Ditaa: Graphical but based on ascii diagrams.
- PlantUML: Includes full breadth of UML options, everything: sequence, use case, class, activity, component, state, and object.

[291] https://github.com/magnars/dash.el
[292] https://github.com/rejeep/f.el
[293] https://github.com/magnars/s.el
[294] http://www.emacswiki.org/emacs/PatternMatching
[295] https://github.com/sindikat/unidecode
[296] http://www.figlet.org/
[297] https://bitbucket.org/jpkotta/figlet
[298] http://www.emacswiki.org/emacs/ArtistMode
[299] http://www.graphviz.org/
[300] http://ditaa.sourceforge.net/
[301] http://plantuml.sourceforge.net/

ditaa was my first pick for usage for the blaring simplicity and power of it. org-mode provides a setup [302]document that I followed. It required installing a JRE and that was about it. artist-mode is automatically loaded for the source block. The example below; thank you [303] Craig Larman

```
+----------+1       1...3+------+
| Sailboat +----------+ Mast |
+-----+----+           +------+
      1
      |
      |              1...2+------+
      +----------------+ Hull |
                       +------+


    This is not a sailboat
```



This [304] ascii-art-to-unicode tool also is interesting.

i. Setup

```
(let ((ditaa-jar "~/java/jar/ditaa0_9.jar"))
  (when (not (file-exists-p ditaa-jar))
    (warn (concat "Can't seem to find a ditaa runtime file where it was "
                  "expected at: " ditaa-jar
                  ". Download a copy here: http://sourceforge.net/projects/ditaa/
  (setq org-ditaa-jar-path ditaa-jar))
```

(b) Org Mode

Late into the development process I ran into some export to HTML issues. After tracking down the source, I learned that tracking down the source of the issue in the source itelf was out of scope for me. My solution was to break out the org-mode configuration into its own block so

[302] http://orgmode.org/worg/org-contrib/babel/languages/ob-doc-ditaa.html
[303] http://www.craiglarman.com/wiki/index.php?title=Books_by_Craig_Larman
[304] http://www.gnuvola.org/software/aa2u/

that I could generate two Emacs configuration files. Doing so with a different section was easier, so that is how I did it.

Tangling can take more than a few minutes; so there is some advice to report on it just to me understand what is happening and that it is working. Another way to perform this monitoring would have been to use around advice. What I ran into is that on HTML export around works fine, but on tangling it did not. This is something that I chose not to investigate; instead I reverted the tangling advice to two separate commands and that seems to work fine.

Org mode, like most of Emacs more powerful modes, slowly grows on you, in pleasant and intuitive ways. Pretty soon, you fall in love with it. After using it for 50+ hours or so you start wanting some easier way to navigate then by typing in the commands over and over (doesn't matter how you re-run them). Reading the miscellaneous section, I learned about speed keys [305] , [306]. Wonderful, just wonderful.

Visualizing hierarchical lists in a flat manner has been fine for me. Sometimes though I wanted to depth-based view, but didn't think much more of it. Reading abouto org-mode, I came upon `org-indent-mode` [307], and decided to give it a try for a while because it is kind of easier to read. Both modes are nice, and thus far I'm switching back and force as I feel like it. Once I opened my eyes and learned about `org-hide-leading-stars` though, I really found happiness.

One topic relating to color themes is that of how code should look within a source block in an org file. There was a thread asking about how to make the block coloring "better". It was interesting because it revealed my preference to myself namely that source blocks should be a muted grey in the document because it shouldn't draw much attention, but in the editor of course you get the highlighting that you want. That is really my personal perference, nonethless, `org-src-fontify-natively` is still always an option.

This [308] article is really fascinating in that crams a ton of information into a tiny space. It also is kind of fun to read because it sumultaneously teaches you so many new things, yet at the same time re-teaches or re-educates you about things that you already knew but didn't know that you could or should be using in these additional manners.

`org2blog/wp` is a nice to have and its customization will be used exclusively for Wisdom&Wonder.

The HTML export of org documents has an optional JavaScript supported display [309]. Not sure how I ended up on this, but it is actually a very nice option. The info style view is nice once you read the directions. The foldinger interface is also interesting; I tried out all 3 generation options but didn't find anything that I specifically liked. Perhaps it is a familiarity or comfort level with GNU styled docs or the keybindings.

For this feature to work, it must come *before* any `org` load statements

```
(setq org-list-allow-alphabetical +1)
```

The bulk of the configuration is as you would expect.

```
(require 'org)
(require 'ox-beamer)
(require 'ox-md)
(require 'htmlize)
```

[305] `http://orgmode.org/manual/Speed-keys.html#Speed-keys`
[306] `http://notesyoujustmightwanttosave.blogspot.com/2011/12/org-speed-keys.html`
[307] `http://orgmode.org/manual/Clean-view.html#Clean-view`
[308] `http://home.fnal.gov/~neilsen/notebook/orgExamples/org-examples.html`
[309] `http://orgmode.org/manual/JavaScript-support.html`

```
(setq org-export-coding-system 'utf-8)

(setq org-export-preserve-breaks nil)

;; (require 'org2blog-autoloads)

(defun gcr/org-mode-hook ()
  (fci-mode)
  (gcr/untabify-buffer-hook)
  (local-set-key (kbd "C-1") 'org-narrow-to-subtree)
  (local-set-key (kbd "M-1") 'widen)
  (local-set-key (kbd "C-2") 'org-edit-special)
;;  (org2blog/wp-mode)
  )

(add-hook 'org-mode-hook 'gcr/org-mode-hook)

(defun gcr/org-src-mode-hook ()
  (local-set-key (kbd "C-2") 'org-edit-src-exit))

(add-hook 'org-src-mode-hook 'gcr/org-src-mode-hook)

(setq org-todo-keywords
      '((sequence "TODO" "IN-PROGRESS" "WAITING" "REVIEW" "DONE")))

(org-babel-do-load-languages
 'org-babel-load-languages
 '((css . t)
   (dot . t)
   (ditaa . t)
   (emacs-lisp . t)
   (js . t)
   (latex . t)
   (lilypond . t)
   (makefile . t)
   (org . t)
   (python . t)
   (plantuml . t)
   (R . t)
   (scheme . t)
   (sh . t)))

(setq org-confirm-babel-evaluate nil)

(setq org-babel-use-quick-and-dirty-noweb-expansion nil)

(setq org-startup-with-inline-images (display-graphic-p))

(setq org-export-copy-to-kill-ring nil)
```

```
(setq org-completion-use-ido +1)

(setq org-use-speed-commands +1)

(setq org-confirm-shell-link-function 'y-or-n-p)

(setq org-confirm-elisp-link-function 'y-or-n-p)

(setq org-enforce-todo-dependencies +1)

(gcr/on-gui
 (require 'org-mouse))

(setq org-pretty-entities +1)

(setq org-ellipsis "...")

(setq org-hide-leading-stars +1)

(setq org-src-fontify-natively nil)

(setq org-fontify-emphasized-text +1)

(setq org-src-preserve-indentation +1)

(setq org-edit-src-content-indentation 0)

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

(mapc (lambda (asc)
        (let ((org-sce-dc (downcase (nth 1 asc))))
          (setf (nth 1 asc) org-sce-dc)))
      org-structure-template-alist)

(when (not (version= (org-version) "8.2.7a"))
  (display-warning
   'org-mode
   (concat
    "Insufficient requirements. Expected 8.2.7a. Found " (org-version))
   :emergency))

(defadvice org-babel-tangle (before org-babel-tangle-before activate)
  (gcr/save-all-file-buffers)
  (message (concat "org-babel-tangle BEFORE: <"
                   (format-time-string "%Y-%m-%dT%T%z")
                   ">")))

(defadvice org-babel-tangle (after org-babel-tangle-after activate)
  (message (concat "org-babel-tangle AFTER: <"
                   (format-time-string "%Y-%m-%dT%T%z")
```

```
                  ">"))
  (alert "Your tangling is complete." :title "org-mode"))

(defadvice org-ascii-export-as-ascii (before org-ascii-export-as-ascii-before activat
  (gcr/save-all-file-buffers))

(defadvice org-html-export-to-html (before before-org-html-export-to-html activate)
  (gcr/save-all-file-buffers)
  (message (concat "org-html-export-to-html BEFORE: <"
                   (format-time-string "%Y-%m-%dT%T%z")
                   ">")))

(defadvice org-html-export-to-html (after after-org-html-export-to-html activate)
  (message (concat "org-html-export-to-html AFTER: <"
                   (format-time-string "%Y-%m-%dT%T%z")
                   ">")))

(when (boundp 'org-html-checkbox-type)
  (display-warning
   'org-mode
   "Org mode now supports HTML export to unicode checkboxes. Please update your confi
   :warning))

(defadvice org-latex-export-to-pdf (before org-latex-export-to-pdf-before activate)
  (gcr/save-all-file-buffers))

(defun sacha/org-html-checkbox (checkbox)
  "Format CHECKBOX into HTML. http://sachachua.com/blog/2014/03/emacs-tweaks-export-o
  (case checkbox (on "<span class=\"check\">&#x2611;</span>") ; checkbox (checked)
        (off "<span class=\"checkbox\">&#x2610;</span>")
        (trans "<code>[-]</code>")
        (t "")))

(defadvice org-html-checkbox (around sacha activate)
  (setq ad-return-value (sacha/org-html-checkbox (ad-get-arg 0))))

(setq org2blog/wp-blog-alist
      '(("wisdomandwonder"
         :url "http://www.wisdomandwonder.com/wordpress/xmlrpc.php"
         :username "admin"
         :default-title "Title goes here"
         :default-categories ("Article")
         :tags-as-categories nil
         :confirm t
         :show 'show
         :keep-new-lines nil
         :wp-latex t
         :wp-code nil
         :track-posts (list "~/wnw.org2blog.org" "Posts"))))
```

There is always a question of how to instill traceability in your artifacts. org provides `:comments` [310] for that. Tangling with that value set to `link`, for example, would add a prefix and postfix comment to the tangled file with the name of the header from which the generated file was tangled. When I tangle the .emacs.el, then it puts something like this for that:

```
;; [[file:~/git/bitbucket-grettke/home/TC3F.org::*Fully%20Loaded%20System][Fully\
file contents go here
;; Fully\ Loaded\ System:1 ends here
```

When you follow the link, it will take you right back to the block that specified the tangling of the document. That is a start, though not super for tracking down details of where the code snippets really originated down to the source blocks themselves.

Trying to understand the other settings, I found `both` to look like this:

```
;; Fully Loaded System ;; Convert decisions into a runnable system.
;; [[file:~/git/bitbucket-grettke/home/TC3F.org::*Fully%20Loaded%20System][Fully\
file contents go here
;; Fully\ Loaded\ System:1 ends here
```

`noweb` looks like, well I'm going to put a couple examples, because this is the best setting. This provides was 99% of org mode literate programmers want which is traceability back from every tangled piece of code to the original document.

```
;; nil ;; nil ;; nil
```

After all of this research, I found that doing noweb-ref tangling, the source locations are not included, so it is no very useful to include comments, and I removed them, at least for now. I am not sure how I want to use them right now.

The type of information that you provide as meta-data is up to you and depends upon your mental model for your org document. My mental model is mostly to use headings as the logical area for addressing a particular *concern* satisfied by that portion of my Emacs configuration, so the tangling comments reflect that. In other words, in this document at least, I rarely name source blocks because the header name is the "true name", and closing the tangle comment with the source block name is really confusing because it usually is `nil`. The org links are fine, too, because they convery all of the necessary information whether you are using org or not. It is more likely that mot readers will not use org links, so they come second.

```
(setq org-babel-tangle-comment-format-beg "line %start-line in %file\n[[%link][%start
(setq org-babel-tangle-comment-format-end (make-string 77 ?=))
```

Probably someone will show be a better way to do repeated string creation :).

Then I read this [311]. **WOW**. That is worth another millions bucks for Eric's prolific contribution to humanity with org-babel.

There is an auto-complete provider for org-mode [312]. Nice as I didn't even think to check. Perhaps a check should go on the standard setup list. This seems to work when you type out things like block definitions; and it won't apply to EasyTemplate generated regions. `auto-complete` will still work on them, though:

```
(require 'org-ac)
(org-ac/config-default)
```

[310] `http://orgmode.org/manual/comments.html#comments`
[311] `http://comments.gmane.org/gmane.emacs.orgmode/32814`
[312] `https://github.com/aki2o/org-ac`

There is a performance issue with tangling when header property inheritance is enabled. Eric explained that [313] there may be performance gains if some of the header properties are not considered. The list below defines what will be allowed, and everything else will be removed.:

```
(let* ((allowed '(exports
                  file
                  noweb
                  noweb-ref
                  session
                  tangle))
       (new-ls
        (--filter (member (car it) allowed)
                  org-babel-common-header-args-w-values)))
  (setq org-babel-common-header-args-w-values new-ls))
```

Footnote management is an important topic. Though I toched on it above, I should expand on that because I kind of forgot a bit, too. First, only define footnotes in-line, it is the only way to support refactoring. Second, randomly generate footnes because it reduces the likelihood of name-collisons occuring during refactoring from one document to another. Those two things are so, so huge:

```
(setq org-footnote-define-inline +1)
(setq org-footnote-auto-label 'random)
(setq org-footnote-auto-adjust nil)
(setq org-footnote-section nil)
```

This is an amazingly easy way to screw up your document. The more you edit org docs, the more you realize how you must truly protect it:

```
(setq org-catch-invisible-edits 'error)
```

Though I am not deliving deep, it is hard not to want to customize some stuff and perhaps this is the start:

```
(setq org-loop-over-headlines-in-active-region t)
```

Doing literate programming a *lot*... it is a hassle to spell-check source blocks, so don't:

```
(add-to-list 'ispell-skip-region-alist '("#\\+begin_src". "#\\+end_src"))
```

4 Assembly

4.1 Prerequisites

4.1.1 Runtime

The entirety of this system is configured for a particular version of Emacs running on Linux and it is not worth fooling around if we aren't running there. It is important enough to at least check and notify the user if those requirements are not met, but not serious enough to kill the editor, and the user ought to have a chance at knowing what is going on.

[313] https://lists.gnu.org/archive/html/emacs-orgmode/2014-06/msg00719.html

```
(when (or
       (not (= emacs-major-version 24))
       (not (= emacs-minor-version 3)))
  (display-warning
   'platform
   (concat
    "Insufficient requirements. Expected v24.3. Found v"
    (number-to-string emacs-major-version) "."
    (number-to-string emacs-minor-version) ".")
   :emergency))
```

4.1.2 Cask [314]

Install required packages first using Cask [315]. Recently I changed the
layout, assuming that order matters for repositories. Logically order would matter too for packages... I'm going to keep my eye open for any issues.

```
(source org)

(depends-on "org-plus-contrib")

(source gnu)

(depends-on "rainbow-mode")

(depends-on "ascii-art-to-unicode")

(source marmalade)

(depends-on "real-auto-save")

(source melpa)

(depends-on "ace-jump-mode")

(depends-on "ace-link")

(depends-on "ace-window")

(depends-on "alert")

(depends-on "auto-complete")

(depends-on "auto-complete-chunk")

(depends-on "boxquote")

(depends-on "dash")
```

[314] https://github.com/cask/cask
[315] https://www.gnu.org/software/emacs/manual/html_node/emacs/Packages.html

```
(depends-on "dash-functional")

(depends-on "diminish")

(depends-on "dired-details+")

(depends-on "erc-hl-nicks")

(depends-on "erc-image")

(depends-on "erc-tweet")

(depends-on "erc-youtube")

(depends-on "ess")

(depends-on "ess-R-data-view")

(depends-on "ess-R-object-popup")

(depends-on "exec-path-from-shell")

(depends-on "expand-region")

(depends-on "f")

(depends-on "fancy-narrow")

(depends-on "figlet")

(depends-on "fill-column-indicator")

(depends-on "flx-ido")

(depends-on "flycheck")

(depends-on "fuzzy")

(depends-on "geiser")

(depends-on "graphviz-dot-mode")

(depends-on "hideshow-org")

(depends-on "diff-hl")

(depends-on "htmlize")

(depends-on "ido-hacks")
```

```
(depends-on "ido-ubiquitous")

(depends-on "ido-vertical-mode")

(depends-on "imenu-anywhere")

(depends-on "json-reformat")

(depends-on "key-chord")

(depends-on "lexbind-mode")

(depends-on "magit")

(depends-on "markdown-mode")

(depends-on "multiple-cursors")

(depends-on "org-ac")

(depends-on "org2blog")

(depends-on "osx-browse")

(depends-on "popup")

(depends-on "pos-tip")

(depends-on "pretty-mode")

(depends-on "projectile")

(depends-on "r-autoyas")

(depends-on "s")

(depends-on "smartparens")

(depends-on "smex")

(depends-on "solarized-theme")

(depends-on "sublimity")

(depends-on "undo-tree")

(depends-on "vagrant")

(depends-on "vagrant-tramp")
```

```
(depends-on "web-mode")

(depends-on "world-time-mode")

(depends-on "xml-rpc")

(depends-on "yasnippet")

;; Local Variables:
;; mode: emacs-lisp
;; End:
```

Tell Emacs how to use Cask.

```
(let ((cask-runtime "~/.cask/cask.el"))
  (when (not (file-exists-p cask-runtime))
    (warn (concat "Can't seem to find a Cask runtime file where it was expected "
                  "at: " cask-runtime " .")))
  (require 'cask cask-runtime))
(defconst gcr/cask-bundle (cask-initialize))
```

4.2 Layout

4.2.1 Detail

Two systems will be configured here:

"Org Only" bare minimum necessary to run org-mode

- Just enough to provide a usable environment
- Both interactively (console, GUI) and non-interactively (interpreter)
- Includes org and **all** of its dependencies
- Standard artifact management with Cask
- Version safety checks
- Makes bug tracking easier
- Reduces likelihood that packages bork org

"Fully Loaded" includes the kitchen sink, too

- In addition to the above
- Every else used in "daily life"

4.2.2 Org Only System

```
<<runtime-check>>
<<general-stuff-block>>
<<custom-variables>>
<<utility-block>>
<<cask-block>>
```

```
<<environment-block>>
<<keymaps-decision>>
<<diagramming-decision>>
<<modes-application-org-mode-module-decision>>
```

Run it like this: `emacs --no-init-file --load .org-mode.emacs.el`

```
<<base-configuration>>
```

4.2.3 Fully Loaded System

Convert decisions into a runnable system.

```
<<base-configuration>>
<<font-block>>
<<uxo-windows-decision>>
(gcr/on-gui
 <<uxo-frames-decision>>)
<<uxo-buffers-decision>>
<<uxo-modeline-decision>>
<<uxo-mark-region-decision>>
<<uxo-minibuffer-decision>>
<<modes-config-decision>>
<<shells-decision>>
<<modes-application-monolith-decision>>
<<communications-decision>>
<<line-wrapping-decision>>
```

4.3 Font block

```
(gcr/on-gui
  <<font-decision>>
  (defun gcr/font-ok-p ()
    "Is the configured font valid?"
    (interactive)
    (member gcr/font-base (font-family-list)))
  (defun gcr/font-name ()
    "Compute the font name and size string."
    (interactive)
    (let* ((size (number-to-string gcr/font-size))
           (name (concat gcr/font-base "-" size)))
      name))
  (defun gcr/update-font ()
    "Updates the current font given configuration values."
    (interactive)
    (if (gcr/font-ok-p)
        (progn
          (message "Setting font to: %s" (gcr/font-name))
          (set-default-font (gcr/font-name)))
      (message (concat "Your preferred font is not available: " gcr/font-base))))
  (defun gcr/text-scale-increase ()
```

```
    "Increase font size"
    (interactive)
    (setq gcr/font-size (+ gcr/font-size 1))
    (gcr/update-font))
  (defun gcr/text-scale-decrease ()
    "Reduce font size."
    (interactive)
    (when (> gcr/font-size 1)
      (setq gcr/font-size (- gcr/font-size 1))
      (gcr/update-font)))

  (gcr/update-font))
```

4.4 Utility fuctions

```
(defun gcr/insert-timestamp ()
  "Produces and inserts a full ISO 8601 format timestamp."
  (interactive)
  (insert (format-time-string "%Y-%m-%dT%T%z")))

(defun gcr/insert-datestamp ()
  "Produces and inserts a partial ISO 8601 format timestamp."
  (interactive)
  (insert (format-time-string "%Y-%m-%d")))

(defun gcr/comment-or-uncomment ()
  "Comment or uncomment the current line or selection."
  (interactive)
  (cond ((not mark-active) (comment-or-uncomment-region (line-beginning-position)
                                                        (line-end-position)))
        ((< (point) (mark)) (comment-or-uncomment-region (point) (mark)))
        (t (comment-or-uncomment-region (mark) (point)))))

(defun gcr/no-control-m ()
  "Aka dos2unix."
  (interactive)
  (let ((line (line-number-at-pos))
        (column (current-column)))
    (mark-whole-buffer)
    (replace-string "
          " "")
    (goto-line line)
    (move-to-column column)))

(defun gcr/untabify-buffer ()
  "For untabifying the entire buffer."
  (interactive)
  (untabify (point-min) (point-max)))

(defun gcr/untabify-buffer-hook ()
```

```
  "Adds a buffer-local untabify on save hook"
  (interactive)
  (add-hook
   'after-save-hook
   (lambda () (gcr/untabify-buffer))
   nil
   'true))

(defun gcr/disable-tabs ()
  "Disables tabs."
  (setq indent-tabs-mode nil))

(defun gcr/save-all-file-buffers ()
  "Saves every buffer associated with a file."
  (interactive)
  (dolist (buf (buffer-list))
    (with-current-buffer buf
      (when (and (buffer-file-name) (buffer-modified-p))
        (save-buffer)))))

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

(defun gcr/delete-trailing-whitespace ()
  "Apply delete-trailing-whitespace to everything but the current line."
  (interactive)
  (let ((first-part-start (point-min))
        (first-part-end (point-at-bol))
        (second-part-start (point-at-eol))
        (second-part-end (point-max)))
    (delete-trailing-whitespace first-part-start first-part-end)
    (delete-trailing-whitespace second-part-start second-part-end)))

(defun gcr/newline ()
  "Locally binds newline."
  (local-set-key (kbd "RET") 'sp-newline))

(defun gcr/describe-thing-in-popup ()
    "Display help information on the current symbol.

Attribution: URL http://www.emacswiki.org/emacs/PosTip
Attribution: URL http://blog.jenkster.com/2013/12/popup-help-in-emacs-lisp.html"
    (interactive)
    (let* ((thing (symbol-at-point))
           (help-xref-following t)
           (description (with-temp-buffer
                          (help-mode)
                          (help-xref-interned thing)
```

```
                          (buffer-string))))
      (gcr/on-gui (pos-tip-show description nil nil nil 300))
      (gcr/not-on-gui (popup-tip description
                                 :point (point)
                                 :around t
                                 :height 30
                                 :scroll-bar t
                                 :margin t))))

(defun gcr/indent-curly-block (&rest _ignored)
  "Open a new brace or bracket expression, with relevant newlines and indent. Src: https://git
  (newline)
  (indent-according-to-mode)
  (forward-line -1)
  (indent-according-to-mode))

(defmacro gcr/on-gnu/linux (statement &rest statements)
  "Evaluate the enclosed body only when run on GNU/Linux."
  `(when (eq system-type 'gnu/linux)
     ,statement
     ,@statements))

(defmacro gcr/on-osx (statement &rest statements)
  "Evaluate the enclosed body only when run on OSX."
  `(when (eq system-type 'darwin)
     ,statement
     ,@statements))

(defmacro gcr/on-windows (statement &rest statements)
  "Evaluate the enclosed body only when run on Microsoft Windows."
  `(when (eq system-type 'windows-nt)
     ,statement
     ,@statements))

(defmacro gcr/on-gui (statement &rest statements)
  "Evaluate the enclosed body only when run on GUI."
  `(when (display-graphic-p)
     ,statement
     ,@statements))

(defmacro gcr/not-on-gui (statement &rest statements)
  "Evaluate the enclosed body only when run on GUI."
  `(when (not (display-graphic-p))
     ,statement
     ,@statements))

(defun beginning-of-line-dwim ()
  "Toggles between moving point to the first non-whitespace character, and
    the start of the line. Src: http://www.wilfred.me.uk/"
  (interactive)
```

```
  (let ((start-position (point)))
    ;; see if going to the beginning of the line changes our position
    (move-beginning-of-line nil)

    (when (= (point) start-position)
      ;; we're already at the beginning of the line, so go to the
      ;; first non-whitespace character
      (back-to-indentation))))

(defun gcr/smart-open-line ()
  "Insert a new line, indent it, and move the cursor there.

This behavior is different then the typical function bound to return
which may be 'open-line' or 'newline-and-indent'. When you call with
the cursor between ^ and $, the contents of the line to the right of
it will be moved to the newly inserted line. This function will not
do that. The current line is left alone, a new line is inserted, indented,
and the cursor is moved there.

Attribution: URL http://emacsredux.com/blog/2013/03/26/smarter-open-line/"
  (interactive)
  (move-end-of-line nil)
  (newline-and-indent))

(defun gcr/narrow-to-region* (boundary-start boundary-end fun)
  "Edit the current region in a new, cloned, indirect buffer.

  This function is responsible for helping the operator to easily
  manipulate a subset of a buffer's contents within a new buffer. The
  newly created clone buffer is created with 'clone-indirect-buffer',
  so all of its behaviors apply. You may care specifically about the
  fact that the clone is really just a 'view' of the source buffer, so
  actions performed within the source buffer or its clone(s) are
  actually occurring only within the source buffer itself. When the
  dynamic extent of this function is entered, the operator is prompted
  for a function to call to make upon entering the new buffer. The intent
  is to specify the desired mode for the new buffer, for example by
  calling 'scheme-mode', but any function may be called.

  The subset chosen for manipulation is narrowed by
  'narrow-to-region'. When the clone buffer is created, the lines in
  which the start and end of the boundary occur are included at the
  end the new clone buffer name to serve as a reminder for its
  'true source'. The intent is to facilitate going back from the clone
  buffer to the source buffer with knowledge of where it originated.

  BOUNDARY-START and BOUNDARY-END are provided by delegation of this
  function to 'interactive'. FUN is provided interactively by the
  operator via the modeline in the same manner. See Info node
  '(elisp) Eval' for more on why 'funcall' was used here instead of
```

```
  ‘eval’ for calling the selected function.

  Attribution: URL ‘http://demonastery.org/2013/04/emacs-narrow-to-region-indirect/’
  Attribution: URL ‘http://paste.lisp.org/display/135818Attribution’"
  (interactive "*r\naMode name? ")
  (let* ((boundary-start (if (< boundary-start 1) (point-min)
                           boundary-start))
         (boundary-end (if (<= boundary-end boundary-start) (point-max)
                         boundary-end))
         (new-name (concat
                    (buffer-name)
                    ""
                    (number-to-string (line-number-at-pos boundary-start))
                    "_"
                    (number-to-string (line-number-at-pos boundary-end))))
         (buf-name (generate-new-buffer-name new-name))
         (fun (if (fboundp fun) fun
                ’fundamental-mode)))
    (with-current-buffer (clone-indirect-buffer buf-name +1 +1)
      (narrow-to-region boundary-start boundary-end)
      (deactivate-mark)
      (goto-char (point-min))
      (funcall fun))))

(defun gcr/set-org-system-header-arg (property value)
  "Easily set system header arguments in org mode.

PROPERTY is the system-wide value that you would like to modify.

VALUE is the new value you wish to store.

Attribution: URL http://orgmode.org/manual/System_002dwide-header-arguments.html#System_002dwi
"
  (setq org-babel-default-header-args
        (cons (cons property value)
              (assq-delete-all property org-babel-default-header-args))))

(defun gcr/insert-ellipsis ()
  "Insert an ellipsis into the current buffer."
  (interactive)
  (insert "..."))

(defun gcr/insert-noticeable-snip-comment-line ()
  "Insert a noticeable snip comment line (NSCL)."
  (interactive)
  (if (not (bolp))
      (message "I may only insert a NSCL at the beginning of a line.")
    (let ((ncl (make-string 70 ?)))
      (newline)
      (previous-line)
```

```
      (insert ncl)
      (comment-or-uncomment-region (line-beginning-position) (line-end-position)))))

(defun gcr/paste-from-x-clipboard()
  "Intelligently grab clipboard information per OS.

Attribution: URL http://blog.binchen.org/posts/paste-string-from-clipboard-into-minibuffer-in-
  (interactive)
  (shell-command
   (cond
    (*cygwin* "getclip")
    (*is-a-mac* "pbpaste")
    (t "xsel -ob")
    )
   1))
```