# Emacs Cheatsheat

Joshua Branson

May 27, 2016

## 1 Why use Emacs?

For starters, Emacs is freedom respecting software. This is often shortened to "free software", or "libre software". This means that one has the right to:

- 0 run the program for any reason.
- 1 the right to help yourself. This is the right to study the source code and change it.
- 2 The right to help your neighbor. This is the right to distribute unmodified copies of Emacs to your neighbor. You could also even sell this program to your neighbor!
- 3 The right to help your community. This is the right to distribute your modified version of Emacs. You could also sell your modified version!

  So Emacs grants you, the user, many freedoms that other programs lack. Please notice that this is computing freedom. Free software is liberty software, not necessarily gratis software. However, Emacs is Free software, and it can usually be obtained as gratis software.

There are also some practical reasons to use Emacs. Plenty of programmers these days use drastically simple programming editors like notepad, notepad++, nano. These editors are "What you see is what you get" type editors, and they are frustratingly lacking in good features, and one cannot easily configure these editors to do anything that you would like. Emacs gives you the ability to install hundreds, if not thousands of add-ons, and configure **everything** that your text editor does!

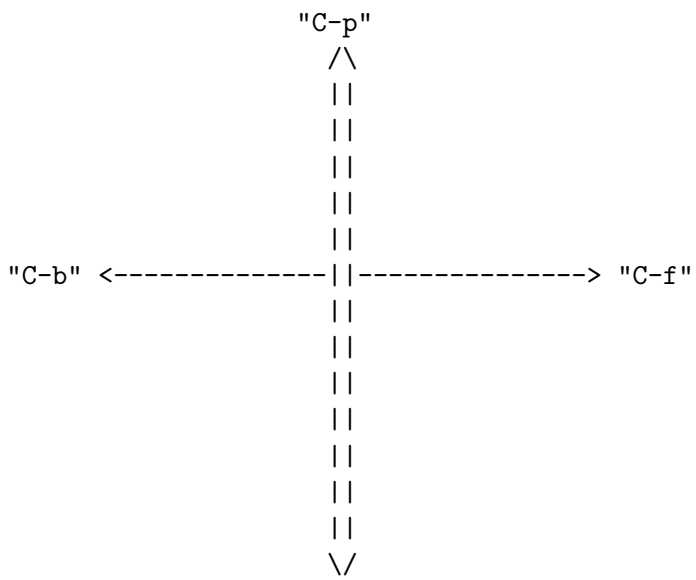
So what can Emacs do? You can use emacs to use commands to manipulate and transpose regions of text, write in nearly every programming

language imaginable, read your email, browse the internet, play games, browse local files, look at a photo gallery, stream internet music, watch online movies (coming soon), create text expandable abbreviations, clock your working hours, invoice clients, schedule your weekly agenda, syntax checking, browse documentation, commit your development changes to version control, emulate vim, all of which can be configured and tweaked emacs in numerous ways. If that doesn't satisfy you, then you can programmatically change emacs' behavior to better suit your development workflow.

# 2 Built in Emacs Features

## 2.1 Movement

In emacs the blinking cursor is called "point". As you type the blinking cursor moves, so we say that point moves as you type. One can always use the arrow keys to move point, but Emacs provides some nifty ways of moving point as well. Pressing and holding the control key whilst hitting "f" will move point forward one character. In Emacs lingo we call that a keychord, and it is denoted like this "C-f". Similarly, holding and pressing the control key, then hitting "b" will move point backward one character. That keychord is denoted as "C-b". "C-p" will move point back to the up previous line, and "C-n" will move point down to the next line. So the basic movement keys look like this:



"C-n"

## 2.2 Bookmarks

Emacs bookmarks are nifty ways of saving your place in a file. If you wish to return to some obscure local or remote file, you can easily save point's current position, and return to it later.

- `C-x r m` sets a bookmark for you at point and it prompts you to name it
- `C-x r b` jumps you to a bookmark

## 2.3 Dired

Dired is the emacs file manager. It opens a buffer displaying all your files in the specified directory. With it you can perform numerous commands on marked files, like deleting, copying, moving, or even creating your own command.

### 2.3.1 Commands

- `n` next line
- `p` previous line
- `m` mark the current file under point
- ~% m REGEXP ¡RET¿ mark files based on a regular expression
- `Q` dired-do-query-replace-regexp lets you search and replace for text inside all marked files
  - Press y, to replace the current match
  - press `n` to skip the current match
  - press `!` to replace all the matches in the current buffer
  - Press `Y` to replace all the remaining matches in all buffers
- how to recursively search and replace
- M-x wdired-change-to-wdired-mode Makes the contents of the dired buffer editable. Once you are done making changes type `C-c C-c`. This saves your changes. It is a nifty way to easily change multiple file permissions, file names, etc.

### 2.3.2 Image Dired

You can also view images inside dired! Mark the images you wish to view, then press C-t d (image-dired-display-thumbs). Alternatively, you could also just run the command M-x image-dired.

## 2.4 Macros

A Macro is a remembered sequence of Emacs keychords that can be repeated. This is useful to easily repeat similar text, or delete, edit similar text. For example, I can write out the numbers 1 through a 100, if I hit 30 keys. I could write the numbers 1 through 1,000 by typing 31 keys!

- `C-x (` begin recording a keyboard macro
- `C-x )` end recording a keyboard macro
- `C-x e` performs the last created keyboard macro
- `C-x q` prompt, when you use this macro if you want to continue. You can enter C-r to briefly step out of the macro and do some edits yourself, then C-M-c will allow you to continue execution of the macro.

## 2.5 Narrowing

Narrowing commands make Emacs only display portions of the buffer, whilst hidding all other regions. While Emacs is narrowed, all entered commands only affected the displayed regions. This means any hidden area cannot be modified while Emacs is narrowed. This is useful if you only want a macro to execute within a specific function. `C-x n <letter> d` narrow to defun `r` widen to region `s` narrow to a org subtree `w` widden to the whole buffer

A much better way to use the narrowing commands is just to make emacs guess what you want whenever you press "C-x n", and that's what the following snippet does. I recommend that you put it in your .emacs:

I found this code snippet here.

```
;; Also set up narrow dwin
(defun narrow-or-widen-dwim (p)
  "Widen if buffer is narrowed, narrow-dwim otherwise.
Dwim means: region, org-src-block, org-subtree, or defun,
whichever applies first. Narrowing to org-src-block actually
calls ‘org-edit-src-code’.
```

```
With prefix P, don't widen, just narrow even if buffer is
already narrowed."
  (interactive "P")
  (declare (interactive-only))
  (cond ((and (buffer-narrowed-p) (not p)) (widen))
        ((region-active-p)
         (narrow-to-region (region-beginning) (region-end)))
        ((derived-mode-p 'org-mode)
         ;; 'org-edit-src-code' is not a real narrowing
         ;; command. Remove this first conditional if you
         ;; don't want it.
         (cond ((ignore-errors (org-edit-src-code))
                (delete-other-windows))
               ((ignore-errors (org-narrow-to-block) t))
               (t (org-narrow-to-subtree))))
        ((derived-mode-p 'latex-mode)
         (LaTeX-narrow-to-environment))
        (t (narrow-to-defun))))

;; This line actually replaces Emacs' entire narrowing
;; keymap, that's how much I like this command. Only copy it
;; if that's what you want.
(define-key ctl-x-map "n" #'narrow-or-widen-dwim)
```

## 2.6 Rectangles

You can easily create a rectangle with evil mode with C-v. Once you have a rectangle you can do these commands:

- `C-x r o` insert blank spaces to the left of the rectangle region
- `C-x r N` insert numbers all along the left of the rectangle region

## 2.7 Registers

Number Registers -'C-u number C-x r n r' Store number into register r (number-to-register).

'C-u number C-x r + r' If r contains a number, increment the number in that register by number. Note that command C-x r + (increment-register) behaves differently if r contains text. See Text Registers.

‘C-x r i r‘ Insert the number from register r into the buffer.

‘C-x r i‘ is the same command used to insert any other sort of register contents into the buffer. ‘C-x r +‘ with no numeric argument increments the register value by 1; ‘C-x r n‘ with no numeric argument stores zero in the register.

`C-x r s R` save text to register R `M-x append-to-register R` appends text to Register R

## 2.8 Windows Commands

In emacs the entire emacs program takes up a **frame**. But emacs allows you to view two different files in the same frame, by splitting the frame in half, or in two **windows**.

- `C-x o` Delete the selected window
- `C-x 1` Delete all the windows except the one that currently has point
- `C-x ^` make the selected window taller
- `C-x` make the selected window narrower
- `C-x` make the selected window wider

## 2.9 org-mode

Emacs org-mode really deserves its own cheatsheat, so I won't go into much detail here, but I'll start you off with the basics. Org-mode is Emacs' organizational mode, and it's pure gold! With Org-mode I organize my daily agenda, todo lists. Parts of my emacs init files is written in it. I use it to keep track of my working hours, with which I then invoice clients. I use its markup to write MIME emails. I wrote all of my documentation in it, and I keep track of my finances with it! It truly is a remarkable emacs mode!

### 2.9.1 Org-mode's hierarchical structure

Org-mode lets you easily insert headings and sub headings with "C-RET". If you press it many times, you'll have something like this:

*
*
*
*

A line with just one "*" is a top level heading. If it has a two "**" below it, then it now has a sub-heading. Just like the following:

```
* I am a top level heading
** I am a sub-heading.
```

**Todo lists** One can easily create simple todo lists with org-mode. In any org file press "C-RET". A "*" will have inserted itself into your buffer. Pressing "C-c C-t" will add the words "TODO", pressing "C-c C-t" again, will change the status to DONE. You will end up with something looking like:

```
* DONE
```

### 2.9.2 org-babel

Org babel is a the best approach towards literate programming ever attempted, and it works! Almost all programming languages treat code as the first order citizen and hides comments behind a simple syntax. For example here is some javascript:

```
// initialize the variable
var i = 5;
if (i < 6) {
  i++;
}
console.log (i);
```

The comment "initialize the variable" comes after the comment syntax "//". In literate programming the code portion of the file is "commented" and the comments do not hide behind a comment syntax. Let me give you an example of the literate kind:

```
Let's write a trivial js function the literate way
#+BEGIN_SRC js :exports code
  var i = 5;
  if (i < 6) {
    i++;
  }
  console.log (i);
#+END_SRC
```

```
5 + 5
```

How cool is that?

**Specific header arguments** `http://orgmode.org/manual/Specific-header-arguments.html org#Specific` header arguments

- :results syntax: :results [raw — silent — value — output ] value is function mode. It means that org-mode will use the last executed command as the value of the output. ie:

```
import time
print("Hello, today's date is %s" % time.ctime())
print('Two plus two is')
return 2 + 2
```

```
echo "hello world"
echo "big cat"
ls -lh | grep emacs.org
```

- :exports [code — results — node — both]
- :dir Specify a default directory that the code is to be run in :dir ¡dir¿

# 3 Helpful Emacs modes

## 3.1 Bug Hunter

This will help you find bugs in your init file. SOOO helpful. `https://github.com/Malabarba/elisp-bug-hunter` M-x bug-hunter-init-file

## 3.2 Helm Mode

HOLY BLIMEY COW!!! THIS MODE IS AWESOME enable it!

C-c C-f helm-find-files

In this mode typing "~/ manage js$" will display a list of files in my home directory that contain the work 'manage' and end is js

Typing C-l will display the files is the parent directory Typing C-z when point is on a directory, will show the files in that directory

Helm has nth commands. Instead of typing tab to get to the action menu just press C-e for the 2nd action and C-j for the 3rd action. You can

also bind a key to an action menu (define-key helm-map (kbd "¡C-tab¿") 'helm-select-4th-action)

C-c C-u lets you view an image

Apparently helm lets you open files externally in helm-find-files with C-c C-x (not working though)

I can make personal actions on to run on files via eshell WOW!!!!

C-SPC will mark files that you can perform actions on

You can then hit copy files in the action menu, which will let you select a new destionation to copy the files. so cool! You can do the same thing from the action menu with rename, symlink, hardlink

helm has support for etags C-h e will search the tags file the above should create the TAGS file with tags, but it's not working find . -iregex .*\.el$ — xargs etags

check out pcomplete extension `https://github.com/thierryvolpiatto/pcomplete-extension`

#### 3.2.1 commands

I can also write my own helm commands cool! write your own helm extentions `C-c h m` open helm-man-woman `C-c h h g` open helm info gnus `C-c h h r` open the helm-emacs-info `C-c h b` is helm-resume which opens up the last instance that you were on `M-<space>` mark candidate `C-h m` inside a helm window will show you all of helm's keybindings

#### 3.2.2 helm-descbinds

### 3.3 Evil Mode

- ~" ¡letter¿ yy~ stores the whole line into register ¡letter¿
- ~" ¡letter¿ p~ puts the whole line back
- `vip` select the paragraph that point is in.

#### 3.3.1 Filters

Filters are VERY cool. You can put the contents of the buffer through a shell command!

You can take a line like

hello how are you !wc -w –¿ 4 hello how are you !wc -c –¿ 21

5 / 6 :.!bc -l –¿ .83333333333333333333

5 :105,109!sort –¿ 1 4 2 3 3 2 4 1 5

Sort has lots of options. I can do sort -r by reverse order, sort -k4 the 4th column, sort -n sort numerically

## 3.4 El-doc

El-doc shows you what a function in the mini-bar as you write it. By default it works for emacs lisp extremely well. You'll notice that my emacs-lisp-mode-hook sharp quotes eldoc-mode, which means it's using the syntax `#'eldoc-mode`. Sharping quoting is only necessary if you are quoting named emacs lisp functions.

```
(add-hook 'emacs-lisp-mode-hook #'eldoc-mode)
```

## 3.5 Semantic

Most emacs modes use a bunch of regular expressions to highlight source code. BUT semantic tries to make this better by parsing the code and creating grammar with it.

### 3.5.1 User commands

- `C-c , j` prompt for a tag in the currect file and move point to it.
- `C-c , J` prompt for a tag in any file that emacs has parsed and move point to it.

## 3.6 Yasnippet

`http://ergoemacs.org/emacs/yasnippet_templates_howto.html`

### 3.6.1 Important Characters

- `$&` indents the line according to the major mode
- `` `(some-lisp-code)` `` embods lisp code
- `$0` where point will be when the snippet ends
- `$n` where n is a number ie: `$1`, `$2`, etc. If you have multiple $3, then typing some text in one $3 will also be put in the other $3.
- =${n:¡placeholder text¿}

## 3.7 Undo Tree

Undo tree is a mode that lets you visually step through the changes that you have done to the buffer. You can step backwards and forwards through time. In normal creation of a documentent, a user typically creates several changes that the emacs undo command is not sufficient to solve. A document's historical content is not always linear. Instead, during normal editing, a user can write content that the normal emacs undo command forgets about. This is where undo tree is helpful. Invoking `M-x undo-tree` shows the user a visual representation of the buffer in time. Using the arrow keys (or conventional emacs replacements), one can step through a documentent's progression.

## 3.8 Paredit Mode

Paredit mode is a superior way to interact with lisps. Since Emacs is configured in emacs lisp, using paredit can be superbly helpful. Paredit attempts to always keep your parentheses balanced. Since lisp is based on parenthesis, this is super helpful. You can find a great introduction to paredit here.

### 3.8.1 slurping

slurping elongates the current sexp by pulling in the closest sexp (either forward or backward)

```
;; here I've called "C-c 0" paredit-forward-slurp-sexp
;; point was always on the "h" in hello
((hello) this is a nice little sentence)
((hello this) is a nice little sentence)
((hello this is) a nice little sentence)
((hello this is a) nice little sentence)
((hello this is a nice) little sentence)
((hello this is a nice little) sentence)
((hello this is a nice little sentence))

;; here I've called C-c 0 paredit-backward-slurp-sexp
;; which pulls in another sexp or atom into my current sexp
;; point was always on "e" in sentence
(hello this is a nice little (sentence))
(hello this is a nice (little sentence))
(hello this is a (nice little sentence))
```

```
(hello this is (a nice little sentence))
(hello this (is a nice little sentence))
(hello (this is a nice little sentence))
((hello this is a nice little sentence))
```

### 3.8.2 barfing

barfing shortens the currect sexp by pushing out the closest sexp (either forward or backward)

```
;; here I've called "C-c ]" paredit-forward-barf-sexp
;; point was always at the "h" in hello
((hello this is a nice little sentence))
((hello this is a nice little) sentence)
((hello this is a nice) little sentence)
((hello this is a) nice little sentence)
((hello) this is a nice little sentence)


;; here I've called paredit-backward-barf-sexp
;; point was always at the "e" in sentence
((hello this is a nice little sentence))
(hello (this is a nice little sentence))
(hello this (is a nice little sentence))
(hello this is (a nice little sentence))
(hello this is a (nice little sentence))
(hello this is a nice (little sentence))
(hello this is a nice little (sentence))
(hello this is a nice little sentence())
```

Paredit

### 3.8.3 paredit-splice-sexp

This removes the parentheses around the current sexpression

```
(cool (cool (cool (cool (cool)))))
(cool (cool (cool (cool (cool)))))
(cool (cool (cool (cool (cool)))))
(cool (cool (cool (cool (cool)))))
(cool (cool (cool (cool (cool)))))
```

## 3.9 Ediff

Ediff is emacs's cool way of comparing two files and merging them into one. `M-x ediff` starts the process. Emacs will prompt you to ediff two files, and then you can begin merging the files together.

### 3.9.1 Commands

- `a` copies buffer a diff to buffer b
- `b` copies buffer b diff to buffer a
- `A` toggles readonly mode of buffer a
- `B` toggles readonly mode of buffer b
- `wa` save buffer a
- `wb` save buffer b
- `!` update the differance regions. If you press `a` and `b` multiple times, you should probably do a `!`
- `*` highlights the words in the diff region that differ
- `ra` restore the diff region in buffer a
- `rb` restore the diff region in buffer b
- `z` suspend the ediff session
- `s` make the merge buffer as small as possible

When you specify files, you can edit the files as root using tramp's syntax like this.

/su::/path/to/file

## 3.10 Tramp

Tramp is an emacs extension that lets you edit remote files. To use tramp, just begin by opening a file via `C-x C-f` (find-file) then typing one of the following special syntaxes:

/HOST:FILENAME /USER@HOST:FILENAME /USER@HOST#PORT:FILENAME /METHOD:USER@HOST:FILENAME /METHOD:USER@HOST#PORT:FILENAME

# 4 Regexp

Regular expressions are nifty ways of searching/replacing regions of text.

Consider this example

```
if (isadmin() || ismanager ()) {
    //some code here
}
```

Suppose that you want to add a space between both "is" in the functions. The following would do this:

`M-x dired-do-query-replace-regexp is(admin|manager) RET is 1 RET`

But let's get a basic understanding of regexps.

# 5 Useful Elisp Libraries

- ctable `https://github.com/kiwanami/emacs-ctable`
- s `https://github.com/magnars/s.el`
- f `https://github.com/rejeep/f.el`
- dash `https://github.com/magnars/dash.el`