Emacs configuration file

Lars Tveito

June 18, 2014

Contents

1 About

This is a Emacs configuration file written in `org-mode`. There are a few reasons why I wanted to do this. My `.emacs.d/` was a mess, and needed a proper clean-up. Also I like keeping all my configurations in a single file, using `org-mode` I can keep this file *organized*. I aim to briefly explain all my configurations.

2 Configurations

2.1 Meta

Emacs can only load `.el`-files. We can use `C-c C-v t` to run `org-babel-tangle`, which extracts the code blocks from the current file into a source-specific file (in this case a `.el`-file).

To avoid doing this each time a change is made we can add a function to the `after-save-hook` ensuring to always tangle and byte-compile the `org`-document after changes.

```
(defun tangle-init ()
  "If the current buffer is 'init.org' the code-blocks are
tangled, and the tangled file is compiled."
  (when (equal (buffer-file-name)
               (expand-file-name (concat user-emacs-directory "init.org")))
    (org-babel-tangle)
    (byte-compile-file (concat user-emacs-directory "init.el"))))

(add-hook 'after-save-hook 'tangle-init)
```

2.2 Package

Managing extensions for Emacs is simplified using `package` which is built in to Emacs 24 and newer. To load downloaded packages we need to initialize `package`.

```
(require 'package)
(setq package-enable-at-startup nil)
(package-initialize)
```

Packages can be fetched from different mirrors, melpa is the largest archive and is well maintained.

```
(setq package-archives
      '(("gnu" . "http://elpa.gnu.org/packages/")
        ("org" . "http://orgmode.org/elpa/")
        ("MELPA" . "http://melpa.milkbox.net/packages/")))
```

We can define a predicate that tells us whether or not the newest version of a package is installed.

```
(defun newest-package-installed-p (package)
  "Return true if the newest available PACKAGE is installed."
  (when (package-installed-p package)
    (let* ((local-pkg-desc (or (assq package package-alist)
                               (assq package package--builtins)))
           (newest-pkg-desc (assq package package-archive-contents)))
      (and local-pkg-desc newest-pkg-desc
           (version-list-= (package-desc-vers (cdr local-pkg-desc))
                           (package-desc-vers (cdr newest-pkg-desc)))))))
```

Let's write a function to install a package if it is not installed or upgrades it if a new version has been released. Here our predicate comes in handy.

```
(defun upgrade-or-install-package (package)
  "Unless the newest available version of PACKAGE is installed
PACKAGE is installed and the current version is deleted."
  (unless (newest-package-installed-p package)
    (let ((pkg-desc (assq package package-alist)))
      (when pkg-desc
        (package-delete (symbol-name package)
                        (package-version-join
                         (package-desc-vers (cdr pkg-desc)))))
      (and (assq package package-archive-contents)
           (package-install package)))))
```

Also, we will need a function to find all dependencies from a given package.

```
(defun dependencies (package)
  "Returns a list of dependencies from a given PACKAGE."
  (let* ((pkg-desc (assq package package-alist))
         (reqs (and pkg-desc (package-desc-reqs (cdr pkg-desc)))))
    (mapcar 'car reqs)))
```

The package-refresh-contents function downloads archive descriptions, this is a major bottleneck in this configuration. To avoid this we can try to only check for updates once every day or so. Here are three variables. The first specifies how often we should check for updates. The second specifies whether one should update during the initialization. The third is a path to a file where a time-stamp is stored in order to check when packages were updated last.

```
(defvar days-between-updates 7)
(defvar do-package-update-on-init t)
(defvar package-last-update-file
  (expand-file-name (concat user-emacs-directory ".package-last-update")))
```

The tricky part is figuring out when packages were last updated. Here is a hacky way of doing it, using time-stamps. By adding a time-stamp to the a file, we can determine whether or not to do an update. After that we must run the time-stamp-function to update the time-stamp.

```
(require 'time-stamp)
;; Open the package-last-update-file
(with-temp-file package-last-update-file
  (if (file-exists-p package-last-update-file)
      (progn
        ;; Insert it's original content's.
        (insert-file-contents package-last-update-file)
        (let ((start (re-search-forward time-stamp-start nil t))
              (end (re-search-forward time-stamp-end nil t)))
          (when (and start end)
            ;; Assuming we have found a time-stamp, we check determine if it's
            ;; time to update.
            (setq do-package-update-on-init
                  (<= days-between-updates
                      (days-between
                       (current-time-string)
                       (buffer-substring-no-properties start end))))
            ;; Remember to update the time-stamp.
            (when do-package-update-on-init
              (time-stamp)))))
    ;; If no such file exists it is created with a time-stamp.
    (insert "Time-stamp: <>")
    (time-stamp)))
```

Now we can use the function above to make sure packages are installed and up to date. Here are some packages I find useful (some of these configurations are also dependent on them).

```
(when (and do-package-update-on-init
           (y-or-n-p "Update all packages?"))
  (package-refresh-contents)

  (let* ((packages
          '(ac-geiser         ; Auto-complete backend for geiser
            ac-slime          ; An auto-complete source using slime completions
            ace-jump-mode     ; quick cursor location minor mode
            auto-compile      ; automatically compile Emacs Lisp libraries
            auto-complete     ; auto completion
            elscreen          ; window session manager
            expand-region     ; Increase selected region by semantic units
            flx-ido           ; flx integration for ido
            ido-vertical-mode ; Makes ido-mode display vertically.
            geiser            ; GNU Emacs and Scheme talk to each other
            haskell-mode      ; A Haskell editing mode
            jedi              ; Python auto-completion for Emacs
            js2-mode          ; Improved JavaScript editing mode
            magit             ; control Git from Emacs
            markdown-mode     ; Emacs Major mode for Markdown-formatted files.
            matlab-mode       ; MATLAB integration with Emacs.
            monokai-theme     ; A fruity color theme for Emacs.
            move-text         ; Move current line or region with M-up or M-down
            multiple-cursors  ; Multiple cursors for Emacs.
            org               ; Outline-based notes management and organizer
            paredit           ; minor mode for editing parentheses
            powerline         ; Rewrite of Powerline
```

```
              pretty-lambdada   ; the word ‘lambda’ as the Greek letter.
              smex              ; M-x interface with Ido-style fuzzy matching.
              undo-tree))       ; Treat undo history as a tree
           ;; Fetch dependencies from all packages.
           (reqs (mapcar 'dependencies packages))
           ;; Append these to the original list, and remove any duplicates.
           (packages (delete-dups (apply 'append packages reqs))))

      (dolist (package packages)
        (upgrade-or-install-package package)))

    ;; This package is only relevant for Mac OS X.
    (when (memq window-system '(mac ns))
      (upgrade-or-install-package 'exec-path-from-shell))
    (package-initialize))
```

2.3 Mac OS X

I run this configuration mostly on Mac OS X, so we need a couple of settings to make things work smoothly. In the package section `exec-path-from-shell` is included (only if you're running OS X), this is to include environment-variables from the shell. It makes useing Emacs along with external processes a lot simpler. I also prefer using the `Command`-key as the `Meta`-key.

```
(when (memq window-system '(mac ns))
  (setq mac-option-modifier nil
        mac-command-modifier 'meta
        x-select-enable-clipboard t)
  (run-with-idle-timer 5 nil 'exec-path-from-shell-initialize))
```

2.4 Require

Some features are not loaded by default to minimize initialization time, so they have to be required (or loaded, if you will). `require`-calls tends to lead to the largest bottleneck's in a configuration. `idle-require` delays the `require`-calls to a time where Emacs is in idle. So this is great for stuff you eventually want to load, but is not a high priority.

```
(require 'idle-require)             ; Need in order to use idle-require
(require 'auto-complete-config)     ; a configuration for auto-complete-mode

(dolist (feature
         '(auto-compile             ; auto-compile .el files
           jedi                     ; auto-completion for python
           matlab                   ; matlab-mode
           ob-matlab                ; org-babel matlab
           ox-latex                 ; the latex-exporter (from org)
           ox-md                    ; Markdown exporter (from org)
           pretty-lambdada          ; show 'lambda' as the greek letter.
           recentf                  ; recently opened files
           smex                     ; M-x interface Ido-style.
           tex-mode))               ; TeX, LaTeX, and SliTeX mode commands
  (idle-require feature))

(setq idle-require-idle-delay 5)
(idle-require-mode 1)
```

2.5 Sane defaults

These are what *I* consider to be saner defaults.

We can set variables to whatever value we'd like using `setq`.

```
(setq default-input-method "TeX"    ; Use TeX when toggeling input method.
      doc-view-continuous t         ; At page edge goto next/previous.
      echo-keystrokes 0.1           ; Show keystrokes asap.
      inhibit-startup-message t     ; No splash screen please.
      initial-scratch-message nil   ; Clean scratch buffer.
      ring-bell-function 'ignore    ; Quiet.
      undo-tree-auto-save-history t ; Save undo history between sessions.
      undo-tree-history-directory-alist
      ;; Put undo-history files in a directory, if it exists.
      (let ((undo-dir (concat user-emacs-directory "undo")))
        (and (file-exists-p undo-dir)
             (list (cons "." undo-dir)))))

;; Some mac-bindings interfere with Emacs bindings.
(when (boundp 'mac-pass-command-to-system)
  (setq mac-pass-command-to-system nil))
```

Some variables are buffer-local, so changing them using `setq` will only change them in a single buffer. Using `setq-default` we change the buffer-local variable's default value.

```
(setq-default fill-column 76                    ; Maximum line width.
              indent-tabs-mode nil              ; Use spaces instead of tabs.
              split-width-threshold 100         ; Split verticly by default.
              auto-fill-function 'do-auto-fill) ; Auto-fill-mode everywhere.
```

The `load-path` specifies where Emacs should look for `.el`-files (or Emacs lisp files). I have a directory called `site-lisp` where I keep all extensions that have been installed manually (these are mostly my own projects).

```
(let ((default-directory (concat user-emacs-directory "site-lisp/")))
  (when (file-exists-p default-directory)
    (normal-top-level-add-to-load-path '("."))
    (normal-top-level-add-subdirs-to-load-path)))
```

Answering *yes* and *no* to each question from Emacs can be tedious, a single *y* or *n* will suffice.

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

To avoid file system clutter we put all auto saved files in a single directory.

```
(defvar emacs-autosave-directory
  (concat user-emacs-directory "autosaves/")
  "This variable dictates where to put auto saves. It is set to a
  directory called autosaves located wherever your .emacs.d/ is
  located.")

;; Sets all files to be backed up and auto saved in a single directory.
(setq backup-directory-alist
      `((".*" . ,emacs-autosave-directory))
      auto-save-file-name-transforms
      `((".*" ,emacs-autosave-directory t)))
```

Set `utf-8` as preferred coding system.

```
(set-language-environment "UTF-8")
```

By default the `narrow-to-region` command is disabled and issues a warning, because it might confuse new users. I find it useful sometimes, and don't want to be warned.

```
(put 'narrow-to-region 'disabled nil)
```

Call `auto-complete` default configuration, which enables `auto-complete` globally.

```
(eval-after-load 'auto-complete-config `(ac-config-default))
```

Automaticly revert `doc-view`-buffers when the file changes on disk.

```
(add-hook 'doc-view-mode-hook 'auto-revert-mode)
```

2.6 Modes

There are some modes that are enabled by default that I don't find particularly useful. We create a list of these modes, and disable all of these.

```
(dolist (mode
         '(tool-bar-mode                ; No toolbars, more room for text.
           scroll-bar-mode              ; No scroll bars either.
           blink-cursor-mode))          ; The blinking cursor gets old.
  (funcall mode 0))
```

Let's apply the same technique for enabling modes that are disabled by default.

```
(dolist (mode
         '(abbrev-mode                 ; E.g. sopl -> System.out.println.
           column-number-mode          ; Show column number in mode line.
           delete-selection-mode       ; Replace selected text.
           recentf-mode                ; Recently opened files.
           show-paren-mode             ; Highlight matching parentheses.
           global-undo-tree-mode))     ; Undo as a tree.
  (funcall mode 1))

(eval-after-load 'auto-compile
  '((auto-compile-on-save-mode 1)))   ; compile .el files on save.
```

This makes `.md`-files open in `markdown-mode`.

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

2.7 Visual

Change the color-theme to monokai (downloaded using package).

```
(load-theme 'monokai t)
```

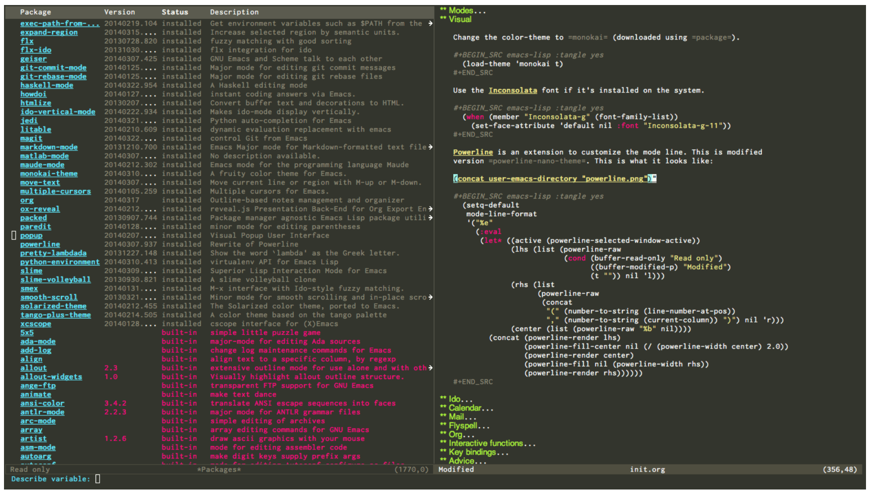
Use the Inconsolata font if it's installed on the system.

```
(when (member "Inconsolata-g" (font-family-list))
  (set-face-attribute 'default nil :font "Inconsolata-g-11"))
```

Powerline is an extension to customize the mode line. This is modified version powerline-nano-theme.

```
(setq-default
 mode-line-format
 '("%e"
   (:eval
    (let* ((active (powerline-selected-window-active))
           ;; left hand side displays Read only or Modified.
           (lhs (list (powerline-raw
                       (cond (buffer-read-only "Read only")
                             ((buffer-modified-p) "Modified")
                             (t "")) nil 'l)))
           ;; right side hand displays (line,column).
           (rhs (list
                 (powerline-raw
                  (concat
                   "(" (number-to-string (line-number-at-pos))
                   "," (number-to-string (current-column)) ")") nil 'r)))
           ;; center displays buffer name.
           (center (list (powerline-raw "%b" nil))))
      (concat (powerline-render lhs)
              (powerline-fill-center nil (/ (powerline-width center) 2.0))
              (powerline-render center)
              (powerline-fill nil (powerline-width rhs))
              (powerline-render rhs))))))
```

This is what it looks like:



```
Package              Version       Status     Description
exec-path-from-... 20140219.104 installed  Get environment variables such as $PATH from the →
expand-region      20140315.... installed  Increase selected region by semantic units.
flx                20130728.820 installed  fuzzy matching with good sorting
flx-ido            20131030.... installed  flx integration for ido
geiser             20140307.425 installed  GNU Emacs and Scheme talk to each other
git-commit-mode    20140125.... installed  Major mode for editing git commit messages
git-rebase-mode    20140125.... installed  Major mode for editing git rebase files
haskell-mode       20140322.954 installed  A Haskell editing mode
howdoi             20140127.... installed  instant coding answers via Emacs.
htmlize            20130207.... installed  Convert buffer text and decorations to HTML.
ido-vertical-mode  20140222.934 installed  Makes ido-mode display vertically.
jedi               20140321.... installed  Python auto-completion for Emacs
litable            20140210.609 installed  dynamic evaluation replacement with emacs
magit              20140322.... installed  control Git from Emacs
markdown-mode      20131210.700 installed  Emacs Major mode for Markdown-formatted text file→
matlab-mode        20140307.... installed  No description available.
maude-mode         20140212.302 installed  Emacs mode for the programming language Maude
monokai-theme      20140310.... installed  A fruity color theme for Emacs.
move-text          20140307.... installed  Move current line or region with M-up or M-down.
multiple-cursors   20140105.259 installed  Multiple cursors for Emacs.
org                20140317     installed  Outline-based notes management and organizer
ox-reveal          20140212.... installed  reveal.js Presentation Back-End for Org Export En→
packed             20130907.744 installed  Package manager agnostic Emacs Lisp package utili→
paredit            20140128.... installed  minor mode for editing parentheses
popup              20140207.... installed  Visual Popup User Interface
powerline          20140307.937 installed  Rewrite of Powerline
pretty-lambdada    20131227.148 installed  Show the word ‘lambda’ as the Greek letter.
python-environment 20140310.413 installed  virtualenv API for Emacs Lisp
slime              20140309.... installed  Superior Lisp Interaction Mode for Emacs
slime-volleyball   20130930.821 installed  A slime volleyball clone
smex               20140131.... installed  M-x interface with Ido-style fuzzy matching.
smooth-scroll      20130321.... installed  Minor mode for smooth scrolling and in-place scro→
solarized-theme    20140212.455 installed  The Solarized color theme, ported to Emacs.
tango-plus-theme   20140214.505 installed  A color theme based on the tango palette
xcscope            20140128.... installed  cscope interface for (X)Emacs
5x5                             built-in   simple little puzzle game
ada-mode                        built-in   major-mode for editing Ada sources
add-log                         built-in   change log maintenance commands for Emacs
align                           built-in   align text to a specific column, by regexp
allout             2.3          built-in   extensive outline mode for use alone and with oth→
allout-widgets     1.0          built-in   Visually highlight allout outline structure.
ange-ftp                        built-in   transparent FTP support for GNU Emacs
animate                         built-in   make text dance
ansi-color         3.4.2        built-in   translate ANSI escape sequences into faces
antlr-mode         2.2.3        built-in   major mode for ANTLR grammar files
arc-mode                        built-in   simple editing of archives
array                           built-in   array editing commands for GNU Emacs
artist             1.2.6        built-in   draw ascii graphics with your mouse
asm-mode                        built-in   mode for editing assembler code
autoarg                         built-in   make digit keys supply prefix args
Read only                     *Packages*                                  (1770,0)
Describe variable:
```

```
** Modes...
** Visual

   Change the color-theme to =monokai= (downloaded using =package=).

   #+BEGIN_SRC emacs-lisp :tangle yes
     (load-theme 'monokai t)
   #+END_SRC

   Use the Inconsolata font if it's installed on the system.

   #+BEGIN_SRC emacs-lisp :tangle yes
     (when (member "Inconsolata-g" (font-family-list))
       (set-face-attribute 'default nil :font "Inconsolata-g-11"))
   #+END_SRC

   Powerline is an extension to customize the mode line. This is modified
   version =powerline-nano-theme=. This is what it looks like:

   (concat user-emacs-directory "powerline.png")"

   #+BEGIN_SRC emacs-lisp :tangle yes
     (setq-default
      mode-line-format
      '("%e"
        (:eval
         (let* ((active (powerline-selected-window-active))
                (lhs (list (powerline-raw
                            (cond (buffer-read-only "Read only")
                                  ((buffer-modified-p) "Modified")
                                  (t "")) nil 'l)))
                (rhs (list
                      (powerline-raw
                       (concat
                        "(" (number-to-string (line-number-at-pos))
                        "," (number-to-string (current-column)) ")") nil 'r)))
                (center (list (powerline-raw "%b" nil))))
           (concat (powerline-render lhs)
                   (powerline-fill-center nil (/ (powerline-width center) 2.0))
                   (powerline-render center)
                   (powerline-fill nil (powerline-width rhs))
                   (powerline-render rhs))))))
   #+END_SRC

** Ido...
** Calendar...
** Mail...
** Flyspell...
** Org...
** Interactive functions...
** Key bindings...
** Advice...
Modified                        init.org                              (356,48)
```

2.8 Ido

Interactive do (or `ido-mode`) changes the way you switch buffers and open files/directories. Instead of writing complete file paths and buffer names you can write a part of it and select one from a list of possibilities. Using `ido-vertical-mode` changes the way possibilities are displayed, and `flx-ido-mode` enables fuzzy matching.

```
(dolist (mode
         '(ido-mode                   ; Interactivly do.
           ido-everywhere             ; Use Ido for all buffer/file reading.
           ido-vertical-mode          ; Makes ido-mode display vertically.
           flx-ido-mode))             ; Toggle flx ido mode.
  (funcall mode 1))
```

We can set the order of file selections in `ido`. I prioritize source files along with `org`- and `tex`-files.

```
(setq ido-file-extensions-order
      '(".el" ".scm" ".lisp" ".java" ".c" ".h" ".org" ".tex"))
```

Sometimes when using `ido-switch-buffer` the `*Messages*` buffer get in the way, so we set it to be ignored (it can be accessed using `C-h e`, so there is really no need for it in the buffer list).

```
(add-to-list 'ido-ignore-buffers "*Messages*")
```

To make M-x behave more like ido-mode we can use the smex package. It needs to be initialized, and we can replace the binding to the standard execute-extended-command with smex.

```
(smex-initialize)
(global-set-key (kbd "M-x") 'smex)
```

2.9 Calendar

Define a function to display week numbers in calender-mode. The snippet is from EmacsWiki.

```
(defun calendar-show-week (arg)
  "Displaying week number in calendar-mode."
  (interactive "P")
  (copy-face font-lock-constant-face 'calendar-iso-week-face)
  (set-face-attribute
   'calendar-iso-week-face nil :height 0.7)
  (setq calendar-intermonth-text
        (and arg
             '(propertize
               (format
                "%2d"
                (car (calendar-iso-from-absolute
                      (calendar-absolute-from-gregorian
                       (list month day year)))))
               'font-lock-face 'calendar-iso-week-face))))
```

Evaluate the calendar-show-week function.

```
(calendar-show-week t)
```

Set Monday as the first day of the week, and set my location.

```
(setq calendar-week-start-day 1
      calendar-latitude 60.0
      calendar-longitude 10.7
      calendar-location-name "Oslo, Norway")
```

2.10 Mail

I use mu4e (which is a part of mu) along with offlineimap on one of my computers. Because the mail-setup wont work without these programs installed we bind `load-mail-setup` to `nil`. If the value is changed to a `non-nil` value mail is setup.

```
(defvar load-mail-setup nil)

(when load-mail-setup
  (eval-after-load 'mu4e
    '(progn
       ;; Some basic mu4e settings.
       (setq mu4e-maildir           "~/.ifimail"     ; top-level Maildir
             mu4e-sent-folder       "/INBOX.Sent"    ; folder for sent messages
             mu4e-drafts-folder     "/INBOX.Drafts"  ; unfinished messages
             mu4e-trash-folder      "/INBOX.Trash"   ; trashed messages
             mu4e-refile-folder     "/INBOX.Archive" ; saved messages
             mu4e-get-mail-command  "offlineimap"    ; offlineimap to fetch mail
             mu4e-compose-signature "- Lars"         ; Sign my name
             mu4e-update-interval   (* 5 60)         ; update every 5 min
             mu4e-confirm-quit      nil              ; just quit
             mu4e-view-show-images  t                ; view images
             mu4e-html2text-command
             "html2text -utf8")                      ; use utf-8

       ;; Setup for sending mail.
       (setq user-full-name
             "Lars Tveito"                        ; Your full name
             user-mail-address
             "larstvei@ifi.uio.no"                ; And email-address
             smtpmail-smtp-server
             "smtp.uio.no"                        ; Host to mail-server
             smtpmail-smtp-service 465            ; Port to mail-server
             smtpmail-stream-type 'ssl            ; Protocol used for sending
             send-mail-function 'smtpmail-send-it ; Use smpt to send
             mail-user-agent 'mu4e-user-agent)    ; Use mu4e!

       ;; Register file types that can be handled by ImageMagick.
       (when (fboundp 'imagemagick-register-types)
        (imagemagick-register-types))))
  (autoload 'mu4e "mu4e" nil t)
  (global-set-key (kbd "C-x m") 'mu4e))
```

2.11 Flyspell

Flyspell offers on-the-fly spell checking. We can enable flyspell for all text-modes with this snippet.

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

To use flyspell for programming there is `flyspell-prog-mode`, that only enables spell checking for comments and strings. We can enable it for all programming modes using the `prog-mode-hook`. Flyspell interferes with auto-complete mode, but there is a workaround provided by auto complete.

```
(add-hook 'prog-mode-hook 'flyspell-prog-mode)
(eval-after-load 'auto-complete
  '(ac-flyspell-workaround))
```

When working with several languages, we should be able to cycle through the languages we most frequently use. Every buffer should have a separate cycle of languages, so that cycling in one buffer does not change the state in a different buffer (this problem occurs if you only have one global cycle). We can implement this by using a closure.

```
(defun cycle-languages ()
  "Changes the ispell dictionary to the first element in
ISPELL-LANGUAGES, and returns an interactive function that cycles
the languages in ISPELL-LANGUAGES when invoked."
  (lexical-let ((ispell-languages '#1=("american" "norsk" . #1#)))
    (ispell-change-dictionary (car ispell-languages))
    (lambda ()
      (interactive)
      ;; Rotates the languages cycle and changes the ispell dictionary.
      (ispell-change-dictionary
       (car (setq ispell-languages (cdr ispell-languages)))))))
```

`Flyspell` signals an error if there is no spell-checking tool is installed. We can advice `turn-on=flyspell` and `flyspell-prog-mode` to only try to enable `flyspell` if a spell-checking tool is available. Also we want to enable cycling the languages by typing `C-c l`, so we bind the function returned from `cycle-languages`.

```
(defadvice turn-on-flyspell (around check nil activate)
  "Turns on flyspell only if a spell-checking tool is installed."
  (when (executable-find ispell-program-name)
    (local-set-key (kbd "C-c l") (cycle-languages))
    ad-do-it))

(defadvice flyspell-prog-mode (around check nil activate)
  "Turns on flyspell only if a spell-checking tool is installed."
  (when (executable-find ispell-program-name)
    (local-set-key (kbd "C-c l") (cycle-languages))
    ad-do-it))
```

2.12 Org

I use `org-agenda` for appointments and such.

```
(setq org-agenda-start-on-weekday nil              ; Show agenda from today.
      org-agenda-files '("~/Dropbox/life.org")     ; A list of agenda files.
      org-agenda-default-appointment-duration 120) ; 2 hours appointments.
```

When editing org-files with source-blocks, we want the source blocks to be themed as they would in their native mode.

```
(setq org-src-fontify-natively t)
```

2.13 Interactive functions

To search recent files useing `ido-mode` we add this snippet from EmacsWiki.

```
(defun recentf-ido-find-file ()
  "Find a recent file using Ido."
  (interactive)
  (let ((f (ido-completing-read "Choose recent file: " recentf-list nil t)))
    (when f
      (find-file f))))
```

`just-one-space` removes all whitespace around a point - giving it a negative argument it removes newlines as well. We wrap a interactive function around it to be able to bind it to a key.

```
(defun remove-whitespace-inbetween ()
  "Removes whitespace before and after the point."
  (interactive)
  (just-one-space -1))
```

This interactive function switches you to a `shell`, and if triggered in the shell it switches back to the previous buffer.

```
(defun switch-to-shell ()
  "Jumps to eshell or back."
  (interactive)
  (if (string= (buffer-name) "*shell*")
      (switch-to-prev-buffer)
    (shell)))
```

To duplicate either selected text or a line we define this interactive function.

```
(defun duplicate-thing ()
  "Ethier duplicates the line or the region"
  (interactive)
  (save-excursion
    (let ((start (if (region-active-p) (region-beginning) (point-at-bol)))
          (end   (if (region-active-p) (region-end) (point-at-eol))))
      (goto-char end)
      (unless (region-active-p)
        (newline))
      (insert (buffer-substring start end)))))
```

To tidy up a buffer we define this function borrowed from simenheg.

```
(defun tidy ()
  "Ident, untabify and unwhitespacify current buffer, or region if active."
  (interactive)
  (let ((beg (if (region-active-p) (region-beginning) (point-min)))
        (end (if (region-active-p) (region-end) (point-max))))
    (indent-region beg end)
    (whitespace-cleanup)
    (untabify beg (if (< end (point-max)) end (point-max)))))
```

Presentation mode.

2.14 Key bindings

Bindings for expand-region.

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

Bindings for multiple-cursors.

```
(global-set-key (kbd "C-c e")  'mc/edit-lines)
(global-set-key (kbd "C-c a")  'mc/mark-all-like-this)
(global-set-key (kbd "C-c n")  'mc/mark-next-like-this)
```

Bindings for Magit.

```
(global-set-key (kbd "C-c m") 'magit-status)
```

Bindings for ace-jump-mode.

```
(global-set-key (kbd "C-c SPC") 'ace-jump-mode)
```

Bindings for `move-text`.

```
(global-set-key (kbd "<M-S-up>")    'move-text-up)
(global-set-key (kbd "<M-S-down>")  'move-text-down)
```

Bind some native Emacs functions.

```
(global-set-key (kbd "C-c s")    'ispell-word)
(global-set-key (kbd "C-c t")    'org-agenda-list)
(global-set-key (kbd "C-x k")    'kill-this-buffer)
(global-set-key (kbd "C-x C-r")  'recentf-ido-find-file)
```

Bind the functions defined above.

```
(global-set-key (kbd "C-c j")    'remove-whitespace-inbetween)
(global-set-key (kbd "C-x t")    'switch-to-shell)
(global-set-key (kbd "C-c d")    'duplicate-thing)
(global-set-key (kbd "<C-tab>")  'tidy)
```

2.15 Advice

An advice can be given to a function to make it behave differently. This advice makes `eval-last-sexp` (bound to `C-x C-e`) replace the sexp with the value.

```
(defadvice eval-last-sexp (around replace-sexp (arg) activate)
  "Replace sexp when called with a prefix argument."
  (if arg
      (let ((pos (point)))
        ad-do-it
        (goto-char pos)
        (backward-kill-sexp)
        (forward-sexp))
    ad-do-it))
```

When interactively changing the theme (using `M-x load-theme`), the current custom theme is not disabled. This often gives weird-looking results; we can advice `load-theme` to always disable themes currently enabled themes.

```
(defadvice load-theme
  (before disable-before-load (theme &optional no-confirm no-enable) activate)
  (mapc 'disable-theme custom-enabled-themes))
```

2.16 Presentation-mode

When giving talks it's nice to be able to scale the text globally. `text-scale-mode` works great for a single buffer, this advice makes this work globally.

```
(defadvice text-scale-mode (around all-buffers (arg) activate)
  (if (not global-text-scale-mode)
      ad-do-it
    (setq-default text-scale-mode-amount text-scale-mode-amount)
    (dolist (buffer (buffer-list))
      (with-current-buffer buffer
        ad-do-it))))
```

We don't want this to be default behavior, so we can make a global mode from the `text-scale-mode`, using `define-globalized-minor-mode`.

```
(require 'face-remap)

(define-globalized-minor-mode
  global-text-scale-mode
  text-scale-mode
  (lambda () (text-scale-mode 1)))
```

3 Language mode specific

3.1 Lisp

`Pretty-lambda` provides a customizable variable `pretty-lambda-auto-modes` that is a list of common lisp modes. Here we can add some extra lisp-modes. We run the `pretty-lambda-for-modes` function to activate `pretty-lambda-mode` in lisp modes.

```
(dolist (mode '(slime-repl-mode geiser-repl-mode))
  (add-to-list 'pretty-lambda-auto-modes mode))

(pretty-lambda-for-modes)
```

I use `Paredit` when editing lisp code, we enable this for all lisp-modes in the `pretty-lambda-auto-modes` list.

```
(dolist (mode pretty-lambda-auto-modes)
  ;; add paredit-mode to all mode-hooks
  (add-hook (intern (concat (symbol-name mode) "-hook")) 'paredit-mode))
```

3.1.1 Emacs Lisp

In `emacs-lisp-mode` we can enable `eldoc-mode` to display information about a function or a variable in the echo area.

```
(add-hook 'emacs-lisp-mode-hook 'turn-on-eldoc-mode)
(add-hook 'lisp-interaction-mode-hook 'turn-on-eldoc-mode)
```

3.1.2 Common lisp

I use Slime along with `lisp-mode` to edit Common Lisp code. Slime provides code evaluation and other great features, a must have for a Common Lisp developer. Quicklisp is a library manager for Common Lisp, and you can install Slime following the instructions from the site along with this snippet.

```
(when (file-exists-p "~/.quicklisp/slime-helper.el")
  (load (expand-file-name "~/.quicklisp/slime-helper.el")))
```

We can specify what Common Lisp program Slime should use (I use SBCL).

```
(setq inferior-lisp-program "sbcl")
```

To improve auto completion for Common Lisp editing we can use `ac-slime` which uses slime completions as a source.

```
(add-hook 'slime-mode-hook 'set-up-slime-ac)
(add-hook 'slime-repl-mode-hook 'set-up-slime-ac)

(eval-after-load "auto-complete"
  '(add-to-list 'ac-modes 'slime-repl-mode))
```

3.1.3 Scheme

Geiser provides features similar to Slime for Scheme editing. Everything works pretty much out of the box, we only need to add auto completion, and specify which scheme-interpreter we prefer.

```
(add-hook 'geiser-mode-hook 'ac-geiser-setup)
(add-hook 'geiser-repl-mode-hook 'ac-geiser-setup)
(eval-after-load "auto-complete"
  '(add-to-list 'ac-modes 'geiser-repl-mode))
(eval-after-load "geiser"
  '(add-to-list 'geiser-active-implementations 'plt-r5rs)) ;'(racket))
```

3.2 Java and C

The `c-mode-common-hook` is a general hook that work on all C-like languages (C, C++, Java, etc...). I like being able to quickly compile using `C-c C-c` (instead of `M-x compile`), a habit from `latex-mode`.

```
(defun c-setup ()
  (local-set-key (kbd "C-c C-c") 'compile))

(require 'auto-complete-c-headers)
(add-to-list 'ac-sources 'ac-source-c-headers)

(add-hook 'c-mode-common-hook 'c-setup)
```

Some statements in Java appear often, and become tedious to write out. We can use abbrevs to speed this up.

```
(define-abbrev-table 'java-mode-abbrev-table
  '(("psv" "public static void main(String[] args) {" nil 0)
    ("sopl" "System.out.println" nil 0)
    ("sop" "System.out.printf" nil 0)))
```

To be able to use the abbrev table defined above, `abbrev-mode` must be activated.

```
(defun java-setup ()
  (abbrev-mode t)
  (setq-local compile-command (concat "javac " (buffer-name))))

(add-hook 'java-mode-hook 'java-setup)
```

3.3 Assembler

When writing assembler code I use # for comments. By defining `comment-start` we can add comments using `M-;` like in other programming modes. Also in assembler should one be able to compile using `C-c C-c`.

```
(defun asm-setup ()
  (setq comment-start "#")
  (local-set-key (kbd "C-c C-c") 'compile))

(add-hook 'asm-mode-hook 'asm-setup)
```

3.4 LaTeX

`.tex`-files should be associated with `latex-mode` instead of `tex-mode`.

```
(add-to-list 'auto-mode-alist '("\\.tex\\'" . latex-mode))
```

I like using the Minted package for source blocks in LaTeX. To make org use this we add the following snippet.

```
(eval-after-load 'org
  '(add-to-list 'org-latex-packages-alist '("" "minted")))
(setq org-latex-listings 'minted)
```

Because Minted uses Pygments (an external process), we must add the `-shell-escape` option to the `org-latex-pdf-process` commands. The `tex-compile-commands` variable controls the default compile command for

Tex- and LaTeX-mode, we can add the flag with a rather dirty statement (if anyone finds a nicer way to do this, please let me know).

```
(eval-after-load 'ox-latex
  '(setq org-latex-pdf-process
         (mapcar
          (lambda (str)
            (concat "pdflatex -shell-escape "
                    (substring str (string-match "-" str))))
          org-latex-pdf-process)))

(eval-after-load 'tex-mode
  '(setcar (cdr (cddaar tex-compile-commands)) " -shell-escape "))
```

3.5 Python

Jedi offers very nice auto completion for `python-mode`. Mind that it is dependent on some python programs as well, so make sure you follow the instructions from the site.

```
;; (setq jedi:server-command
;;       (cons "python3" (cdr jedi:server-command))
;;       python-shell-interpreter "python3")
(add-hook 'python-mode-hook 'jedi:setup)
(setq jedi:complete-on-dot t)
(add-hook 'python-mode-hook 'jedi:ac-setup)
```

3.6 Haskell

`haskell-doc-mode` is similar to `eldoc`, it displays documentation in the echo area. Haskell has several indentation modes - I prefer using `haskell-indent`.

```
(add-hook 'haskell-mode-hook 'turn-on-haskell-doc-mode)
(add-hook 'haskell-mode-hook 'turn-on-haskell-indent)
```

3.7 Matlab

`Matlab-mode` works pretty good out of the box, but we can do without the splash screen.

```
(eval-after-load 'matlab
  '(add-to-list 'matlab-shell-command-switches "-nosplash"))
```