Aaron Bedra's Emacs 26 Configuration

Table of Contents

Configuration

Emacs is a special beast. Taming it takes a lot of care. In an attempt to document/explain /share with the rest of the world, this is my attempt at configuration as a literate program. It also shows off the awesome power of org-mode, which makes all of this possible.

User details

Emacs will normally pick this up automatically, but this way I can be sure the right information is always present.

```
(setq user-full-name "Aaron Bedra")
(setq user-mail-address "aaron@aaronbedra.com")
```

Environment

There are plenty of things installed outside of the default `PATH`. This allows me to establish additional `PATH` information. At the moment, the only things that added are `/usr/local/bin` for homebrew on OS X and `.cabal/bin` for Haskell package binaries.

Emacs lisp is really only a subset of common lisp, and I need to have some of the additional functionality to make the configuration and its dependencies work properly, which we get by requiring Common Lisp for Emacs.

```
(setenv "PATH" (concat "/usr/local/bin:/opt/local/bin:/usr/bin:/bin:
/home/abedra/.cabal/bin" (getenv "PATH")))
(setenv "GOPATH" (concat (getenv "HOME") "/src/golang"))
(setq exec-path (append exec-path '("/usr/local/bin")))
(add-to-list 'exec-path (concat (getenv "GOPATH") "/bin"))
(require 'cl)
```

Package Management

Since Emacs 24, Emacs includes the Emacs Lisp Package Archive (ELPA) by default. This provides a nice way to install additional packages. Since the default package archive doesn't include everything necessary, the marmalade, and melpa repositories are also added.

```
(load "package")
(package-initialize)
(add-to-list 'package-archives
             '("marmalade" . "http://marmalade-repo.org/packages/"))
(add-to-list 'package-archives
             '("melpa" . "http://melpa.milkbox.net/packages/") t)

(setq package-archive-enable-alist '(("melpa" magit f)))
```

Define default packages

This is the list of packages used in this configuration.

```
(defvar abedra/packages '(ac-slime
                          auto-complete
                          autopair
                          cider
                          clojure-mode
                          ess
                          f
                          feature-mode
                          flycheck
                          go-autocomplete
                          go-eldoc
                          go-mode
                          graphviz-dot-mode
                          haml-mode
                          htmlize
                          magit
                          markdown-mode
                          marmalade
                          org
                          paredit
                          powerline
                          rvm
                          smex
                          solarized-theme
                          web-mode
                          writegood-mode
                          yaml-mode)
  "Default packages")
```

Install default packages

When Emacs boots, check to make sure all of the packages defined in `abedra/packages` are installed. If not, have ELPA take care of it.

```
(defun abedra/packages-installed-p ()
  (loop for pkg in abedra/packages
        when (not (package-installed-p pkg)) do (return nil)
        finally (return t)))

(unless (abedra/packages-installed-p)
  (message "%s" "Refreshing package database...")
  (package-refresh-contents)
  (dolist (pkg abedra/packages)
```

```
    (when (not (package-installed-p pkg))
      (package-install pkg))))
```

Start-up options

Splash Screen

I want to skip straight to the scratch buffer. This turns off the splash screen and puts me straight into the scratch buffer. I don't really care to have anything in there either, so turn off the message while we're at it. Since I end up using `org-mode` most of the time, set the default mode accordingly.

```
(setq inhibit-splash-screen t
      initial-scratch-message nil
      initial-major-mode 'org-mode)
```

Scroll bar, Tool bar, Menu bar

Emacs starts up with way too much enabled. Turn off the scroll bar, menu bar, and tool bar. There isn't really a reason to have them on.

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

Marking text

There are some behaviors in Emacs that aren't intuitive. Since I pair with others that don't know how Emacs handles highlighting, treat regions like other text editors. This means typing when the mark is active will write over the marked region. Also, make the common highlighting keystrokes work the way most people expect them to. This saves a lot of time explaining how to highlight areas of text. Emacs also has it's own clipboard and doesn't respond to the system clipboard by default, so tell Emacs that we're all friends and can get along.

```
(delete-selection-mode t)
(transient-mark-mode t)
(setq x-select-enable-clipboard t)
```

Display Settings

I have some modifications to the default display. First, a minor tweak to the frame title. It's also nice to be able to see when a file actually ends. This will put empty line markers into the left hand side.

```
(setq-default indicate-empty-lines t)
(when (not indicate-empty-lines)
  (toggle-indicate-empty-lines))
```

Indentation

There's nothing I dislike more than tabs in my files. Make sure I don't share that discomfort with others.

```
(setq tab-width 2
      indent-tabs-mode nil)
```

Backup files

Some people like to have them. I don't. Rather than pushing them to a folder, never to be used, just turn the whole thing off.

```
(setq make-backup-files nil)
```

Yes and No

Nobody likes to have to type out the full yes or no when Emacs asks. Which it does often. Make it one character.

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

Key bindings

Miscellaneous key binding stuff that doesn't fit anywhere else.

```
(global-set-key (kbd "RET") 'newline-and-indent)
(global-set-key (kbd "C-;") 'comment-or-uncomment-region)
(global-set-key (kbd "M-/") 'hippie-expand)
(global-set-key (kbd "C-+") 'text-scale-increase)
(global-set-key (kbd "C--") 'text-scale-decrease)
(global-set-key (kbd "C-c C-k") 'compile)
(global-set-key (kbd "C-x g") 'magit-status)
```

Misc

Turn down the time to echo keystrokes so I don't have to wait around for things to happen. Dialog boxes are also a bit annoying, so just have Emacs use the echo area for everything. Beeping is for robots, and I am not a robot. Use a visual indicator instead of making horrible noises. Oh, and always highlight parentheses. A person could go insane without that.

```
(setq echo-keystrokes 0.1
      use-dialog-box nil
      visible-bell t)
(show-paren-mode t)
```

Vendor directory

I have a couple of things that don't come from package managers. This includes the directory for use.

```
(defvar abedra/vendor-dir (expand-file-name "vendor" user-emacs-
directory))
(add-to-list 'load-path abedra/vendor-dir)

(dolist (project (directory-files abedra/vendor-dir t "\\w+"))
  (when (file-directory-p project)
    (add-to-list 'load-path project)))
```

Org

`org-mode` is one of the most powerful and amazing features of Emacs. I mostly use it for task/day organization and generating code snippets in HTML. Just a few tweaks here to make the experience better.

Settings

Enable logging when tasks are complete. This puts a time-stamp on the completed task. Since I usually am doing quite a few things at once, I added the `INPROGRESS` keyword and made the color blue. Finally, enable `flyspell-mode` and `writegood-mode` when `org-mode` is active.

```
(setq org-log-done t
      org-todo-keywords '((sequence "TODO" "INPROGRESS" "DONE"))
      org-todo-keyword-faces '(("INPROGRESS" . (:foreground "blue"
:weight bold))))
(add-hook 'org-mode-hook
          (lambda ()
            (flyspell-mode)))
(add-hook 'org-mode-hook
          (lambda ()
            (writegood-mode)))
```

org-agenda

First, create the global binding for `org-agenda`. This allows it to be quickly accessed. The agenda view requires that org files be added to it. The `personal.org` file is my daily file for review. I have a habit to plan the next day. I do this by assessing my calendar and my

list of todo items. If a todo item is already scheduled or has a deadline, don't show it in the global todo list.

```
(global-set-key (kbd "C-c a") 'org-agenda)
(setq org-agenda-show-log t
      org-agenda-todo-ignore-scheduled t
      org-agenda-todo-ignore-deadlines t)
(setq org-agenda-files (list "~/Dropbox/org/personal.org"))
```

org-habit

I have several habits that I also track. In order to take full advantage of this feature `org-habit` has to be required and added to `org-modules`. A few settings are also tweaked for habit mode to make the tracking a little more palatable. The most significant of these is `org-habit-graph-column`. This specifies where the graph should start. The default is too low and cuts off a lot, so I start it at 80 characters.

```
(require 'org)
(require 'org-install)
(require 'org-habit)
(add-to-list 'org-modules "org-habit")
(setq org-habit-preceding-days 7
      org-habit-following-days 1
      org-habit-graph-column 80
      org-habit-show-habits-only-for-today t
      org-habit-show-all-today t)
```

org-babel

`org-babel` is a feature inside of `org-mode` that makes this document possible. It allows for embedding languages inside of an `org-mode` document with all the proper font-locking. It also allows you to extract and execute code. It isn't aware of `Clojure` by default, so the following sets that up.

```
(require 'ob)

(org-babel-do-load-languages
```

```
 'org-babel-load-languages
 '((shell . t)
   (ditaa . t)
   (plantuml . t)
   (dot . t)
   (ruby . t)
   (js . t)
   (C . t)))

(add-to-list 'org-src-lang-modes (quote ("dot". graphviz-dot)))
(add-to-list 'org-src-lang-modes (quote ("plantuml" . fundamental)))
(add-to-list 'org-babel-tangle-lang-exts '("clojure" . "clj"))

(defvar org-babel-default-header-args:clojure
  '((:results . "silent") (:tangle . "yes")))

(defun org-babel-execute:clojure (body params)
  (lisp-eval-string body)
  "Done!")

(provide 'ob-clojure)

(setq org-src-fontify-natively t
      org-confirm-babel-evaluate nil)

(add-hook 'org-babel-after-execute-hook (lambda ()
                                          (condition-case nil
                                              (org-display-inline-
images)
                                            (error nil)))
          'append)
```

org-abbrev

```
(add-hook 'org-mode-hook (lambda () (abbrev-mode 1)))

(define-skeleton skel-org-block-elisp
  "Insert an emacs-lisp block"
  ""
  "#+begin_src emacs-lisp\n"
  _ - \n
  "#+end_src\n")

(define-abbrev org-mode-abbrev-table "elsrc" "" 'skel-org-block-
```

```
elisp)

(define-skeleton skel-org-block-js
  "Insert a JavaScript block"
  ""
  "#+begin_src js\n"
  _ - \n
  "#+end_src\n")

(define-abbrev org-mode-abbrev-table "jssrc" "" 'skel-org-block-js)

(define-skeleton skel-header-block
  "Creates my default header"
  ""
  "#+TITLE: " str "\n"
  "#+AUTHOR: Aaron Bedra\n"
  "#+EMAIL: aaron@aaronbedra.com\n"
  "#+OPTIONS: toc:3 num:nil\n"
  "#+STYLE: <link rel=\"stylesheet\" type=\"text/css\"
href=\"http://thomasf.github.io/solarized-css/solarized-
light.min.css\" />\n")

(define-abbrev org-mode-abbrev-table "sheader" "" 'skel-header-
block)

(define-skeleton skel-org-html-file-name
  "Insert an HTML snippet to reference the file by name"
  ""
  "#+HTML: <strong><i>"str"</i></strong>")

(define-abbrev org-mode-abbrev-table "fname" "" 'skel-org-html-file-
name)

(define-skeleton skel-ngx-config
  "Template for NGINX module config file"
  ""
  "ngx_addon_name=ngx_http_" str  "_module\n"
  "HTTP_MODULES=\"$HTTP_MODULES ngx_http_" str "_module\"\n"
  "NGX_ADDON_SRCS=\"$NGX_ADDON_SRCS $ngx_addon_dir/ngx_http_" str
"_module.c\"")

(define-abbrev fundamental-mode-abbrev-table "ngxcnf" "" 'skel-ngx-
config)

(define-skeleton skel-ngx-module
  "Template for NGINX modules"
  ""
  "#include <nginx.h>\n"
```

```
  "#include <ngx_config.h>\n"
  "#include <ngx_core.h>\n"
  "#include <ngx_http.h>\n\n"

  "ngx_module_t ngx_http_" str "_module;\n\n"

  "static ngx_int_t\n"
  "ngx_http_" str "_handler(ngx_http_request_t *r)\n"
  "{\n"
  >"if (r->main->internal) {\n"
  >"return NGX_DECLINED;\n"
  "}" > \n
  \n
  >"ngx_log_error(NGX_LOG_ERR, r->connection->log, 0, \"My new
module\");\n\n"
  > _ \n
  >"return NGX_OK;\n"
  "}" > "\n\n"

  "static ngx_int_t\n"
  "ngx_http_"str"_init(ngx_conf_t *cf)\n"
  "{\n"
  >"ngx_http_handler_pt *h;\n"
  >"ngx_http_core_main_conf_t *cmcf;\n\n"

  >"cmcf = ngx_http_conf_get_module_main_conf(cf,
ngx_http_core_module);\n"
  >"h =
ngx_array_push(&cmcf->phases[NGX_HTTP_ACCESS_PHASE].handlers);\n\n"

  >"if (h == NULL) {\n"
  >"return NGX_ERROR;\n"
  "}" > \n
  \n
  >"*h = ngx_http_"str"_handler;\n\n"

  >"return NGX_OK;\n"
  "}" > \n
  \n
  "static ngx_http_module_t ngx_http_"str"_module_ctx = {\n"
  >"NULL,                  /* preconfiguration */\n"
  >"ngx_http_"str"_init,   /* postconfiguration */\n"
  >"NULL,                  /* create main configuration */\n"
  >"NULL,                  /* init main configuration */\n"
  >"NULL,                  /* create server configuration */\n"
  >"NULL,                  /* merge server configuration */\n"
  >"NULL,                  /* create location configuration */\n"
  >"NULL                   /* merge location configuration */\n"
```

```
  "};" > \n
  \n

  "ngx_module_t ngx_http_"str"_module = {\n"
  >"NGX_MODULE_V1,\n"
  >"&ngx_http_"str"_module_ctx,  /* module context */\n"
  >"NULL,                        /* module directives */\n"
  >"NGX_HTTP_MODULE,             /* module type */\n"
  >"NULL,                        /* init master */\n"
  >"NULL,                        /* init module */\n"
  >"NULL,                        /* init process */\n"
  >"NULL,                        /* init thread */\n"
  >"NULL,                        /* exit thread */\n"
  >"NULL,                        /* exit process */\n"
  >"NULL,                        /* exit master */\n"
  >"NGX_MODULE_V1_PADDING\n"
  "};" >)

(require 'cc-mode)
(define-abbrev c-mode-abbrev-table "ngxmod" "" 'skel-ngx-module)

(define-skeleton skel-ngx-append-header
  "Template for header appending function for NGINX modules"
  ""
  "static void append_header(ngx_http_request_t *r)\n"
  "{\n"
  > "ngx_table_elt_t *h;\n"
  > "h = ngx_list_push(&r->headers_out.headers);\n"
  > "h->hash = 1;\n"
  > "ngx_str_set(&h->key, \"X-NGINX-Hello\");\n"
  > "ngx_str_set(&h->value, \"Hello NGINX!\");\n"
  "}\n")

(define-abbrev c-mode-abbrev-table "ngxhdr" "" 'skel-ngx-append-
header)
```

Utilities

ditaa

There's no substitute for real drawings, but it's nice to be able to sketch things out and produce a picture right from `org-mode`. This sets up `ditaa` for execution from inside a babel block.

```
(setq org-ditaa-jar-path "~/.emacs.d/vendor/ditaa0_9.jar")
```

plantuml

```
(setq org-plantuml-jar-path "~/.emacs.d/vendor/plantuml.jar")
```

deft

`deft` provides random note taking with history and searching. Since I use `org-mode` for everything else, I turn that on as the default mode for `deft` and put the files in Dropbox.

```
(setq deft-directory "~/Dropbox/deft")
(setq deft-use-filename-as-title t)
(setq deft-extension "org")
(setq deft-text-mode 'org-mode)
```

Smex

`smex` is a necessity. It provides history and searching on top of `M-x`.

```
(setq smex-save-file (expand-file-name ".smex-items" user-emacs-
directory))
(smex-initialize)
(global-set-key (kbd "M-x") 'smex)
(global-set-key (kbd "M-X") 'smex-major-mode-commands)
```

Ido

`Ido` mode provides a nice way to navigate the filesystem. This is mostly just turning it on.

```
(ido-mode t)
```

```
(setq ido-enable-flex-matching t
      ido-use-virtual-buffers t)
```

Column number mode

Turn on column numbers.

```
(setq column-number-mode t)
```

Temporary file management

Deal with temporary files. I don't care about them and this makes them go away.

```
(setq backup-directory-alist `((".*" . ,temporary-file-directory)))
(setq auto-save-file-name-transforms `((".*" ,temporary-file-
directory t)))
```

autopair-mode

This makes sure that brace structures `()`, `[]`, `{}`, etc. are closed as soon as the opening character is typed.

```
(require 'autopair)
```

Power lisp

A bunch of tweaks for programming in LISP dialects. It defines the modes that I want to apply these hooks to. To add more just add them to `lisp-modes`. This also creates its own minor mode to properly capture the behavior. It remaps some keys to make paredit work a little easier as well. It also sets `clisp` as the default lisp program and `racket` as the default scheme program.

```
(setq lisp-modes '(lisp-mode
                   emacs-lisp-mode
                   common-lisp-mode
                   scheme-mode
                   clojure-mode))

(defvar lisp-power-map (make-keymap))
(define-minor-mode lisp-power-mode "Fix keybindings; add power."
  :lighter " (power)"
  :keymap lisp-power-map
  (paredit-mode t))
(define-key lisp-power-map [delete] 'paredit-forward-delete)
(define-key lisp-power-map [backspace] 'paredit-backward-delete)

(defun abedra/engage-lisp-power ()
  (lisp-power-mode t))

(dolist (mode lisp-modes)
  (add-hook (intern (format "%s-hook" mode))
            #'abedra/engage-lisp-power))

(setq inferior-lisp-program "clisp")
(setq scheme-program-name "racket")
```

auto-complete

Turn on auto complete.

```
(require 'auto-complete-config)
(ac-config-default)
```

Indentation and buffer cleanup

This re-indents, untabifies, and cleans up whitespace. It is stolen directly from the emacs-starter-kit.

```
(defun untabify-buffer ()
  (interactive)
  (untabify (point-min) (point-max)))
```

```
(defun indent-buffer ()
  (interactive)
  (indent-region (point-min) (point-max)))

(defun cleanup-buffer ()
  "Perform a bunch of operations on the whitespace content of a
buffer."
  (interactive)
  (indent-buffer)
  (untabify-buffer)
  (delete-trailing-whitespace))

(defun cleanup-region (beg end)
  "Remove tmux artifacts from region."
  (interactive "r")
  (dolist (re '("\\\\ | \·*\n" "\W* | \·*"))
    (replace-regexp re "" nil beg end)))

(global-set-key (kbd "C-x M-t") 'cleanup-region)
(global-set-key (kbd "C-c n") 'cleanup-buffer)

(setq-default show-trailing-whitespace t)
```

flyspell

The built-in Emacs spell checker. Turn off the welcome flag because it is annoying and breaks on quite a few systems. Specify the location of the spell check program so it loads properly.

```
(setq flyspell-issue-welcome-flag nil)
(if (eq system-type 'darwin)
    (setq-default ispell-program-name "/usr/local/bin/aspell")
  (setq-default ispell-program-name "/usr/bin/aspell"))
(setq-default ispell-list-command "list")
```

eshell

Customize eshell

```
(setq eshell-visual-commands
      '("less" "tmux" "htop" "top" "bash" "zsh" "fish"))

(setq eshell-visual-subcommands
      '(("git" "log" "l" "diff" "show")))

;; Prompt with a bit of help from http://www.emacswiki.org/emacs
/EshellPrompt
(defmacro with-face (str &rest properties)
  `(propertize ,str 'face (list ,@properties)))

(defun eshell/abbr-pwd ()
  (let ((home (getenv "HOME"))
        (path (eshell/pwd)))
    (cond
     ((string-equal home path) "~")
     ((f-ancestor-of? home path) (concat "~/" (f-relative path
home)))
     (path))))

(defun eshell/my-prompt ()
  (let ((header-bg "#161616"))
    (concat
;     (with-face user-login-name :foreground "#dc322f")
;     (with-face (concat "@" hostname) :foreground "#268bd2")
;     " "
     (with-face (eshell/abbr-pwd) :foreground "#008700")
     (if (= (user-uid) 0)
         (with-face "#" :foreground "red")
       (with-face "$" :foreground "#2345ba"))
     " ")))

(setq eshell-prompt-function 'eshell/my-prompt)
(setq eshell-highlight-prompt nil)
(setq eshell-prompt-regexp "^[^#$\n]+[#$] ")

(setq eshell-cmpl-cycle-completions nil)
```

powerline

```
(require 'powerline)
(powerline-default-theme)
```

Language Hooks

shell-script-mode

Use `shell-script-mode` for `.zsh` files.

```
(add-to-list 'auto-mode-alist '("\\.zsh$" . shell-script-mode))
```

conf-mode

```
(add-to-list 'auto-mode-alist '("\\.gitconfig$" . conf-mode))
```

Web Mode

```
(setq web-mode-style-padding 2)
(setq web-mode-script-padding 2)
(setq web-mode-markup-indent-offset 2)
(setq web-mode-css-indent-offset 2)
(setq web-mode-code-indent-offset 2)

(add-to-list 'auto-mode-alist '("\\.hbs$" . web-mode))
(add-to-list 'auto-mode-alist '("\\.erb$" . web-mode))
(add-to-list 'auto-mode-alist '("\\.html$" . web-mode))
```

Ruby

Turn on `autopair` for Ruby. Identify additional file names/extensions that will trigger `ruby-mode` when loaded.

```
(add-hook 'ruby-mode-hook
          (lambda ()
            (autopair-mode)))
```

```
(add-to-list 'auto-mode-alist '("\\.rake$" . ruby-mode))
(add-to-list 'auto-mode-alist '("\\.gemspec$" . ruby-mode))
(add-to-list 'auto-mode-alist '("\\.ru$" . ruby-mode))
(add-to-list 'auto-mode-alist '("Rakefile" . ruby-mode))
(add-to-list 'auto-mode-alist '("Gemfile" . ruby-mode))
(add-to-list 'auto-mode-alist '("Capfile" . ruby-mode))
(add-to-list 'auto-mode-alist '("Vagrantfile" . ruby-mode))
(add-to-list 'auto-mode-alist '("Guardfile" . ruby-mode))
```

RVM

Enable Ruby Version Manager mode and tell it to use the default Ruby.

```
;; (rvm-use-default) ;; This is causing a 1.5 second slow down to
startup, disabling for now
```

YAML

Add additional file extensions that trigger `yaml-mode`.

```
(add-to-list 'auto-mode-alist '("\\.yml$" . yaml-mode))
(add-to-list 'auto-mode-alist '("\\.yaml$" . yaml-mode))
```

CoffeeScript Mode

The default CoffeeScript mode makes terrible choices. This turns everything into 2 space indentations and makes it so the mode functions rather than causing you indentation errors every time you modify a file.

```
(defun coffee-custom ()
  "coffee-mode-hook"
  (make-local-variable 'tab-width)
  (set 'tab-width 2))

(add-hook 'coffee-mode-hook 'coffee-custom)
```

JavaScript Mode

`js-mode` defaults to using 4 spaces for indentation. Change it to 2

```
(defun js-custom ()
  "js-mode-hook"
  (setq js-indent-level 2))

(add-hook 'js-mode-hook 'js-custom)
```

Markdown Mode

Enable Markdown mode and setup additional file extensions. Use pandoc to generate HTML previews from within the mode, and use a custom css file to make it a little prettier.

```
(add-to-list 'auto-mode-alist '("\\.md$" . markdown-mode))
(add-to-list 'auto-mode-alist '("\\.mdown$" . markdown-mode))
(add-hook 'markdown-mode-hook
          (lambda ()
            (visual-line-mode t)
            (writegood-mode t)
            (flyspell-mode t)))
(setq markdown-command "pandoc --smart -f markdown -t html")
(setq markdown-css-paths `(,(expand-file-name "markdown.css"
abedra/vendor-dir)))
```

Idris Mode

```
(setq idris-interpreter-path "/usr/local/bin/idris")
```

CPSA Mode

Enable support for Cryptographic Protocol Shapes Analyzer. This is a scheme-ish dialect, so it's a derived from `scheme-mode`.

```
(define-derived-mode cpsa-mode scheme-mode
  (setq mode-name "CPSA")
  (setq cpsa-keywords '("defmacro" "defprotocol" "defrole"
"defskeleton" "defstrand"))
  (setq cpsa-functions '("cat" "hash" "enc" "string" "ltk" "privk"
"pubk" "invk" "send" "recv"  "non-orig" "uniq-orig" "trace" "vars"))
  (setq cpsa-types '("skey" "akey" "name" "text"))
  (setq cpsa-keywords-regexp (regexp-opt cpsa-keywords 'words))
  (setq cpsa-functions-regexp (regexp-opt cpsa-functions 'words))
  (setq cpsa-types-regexp (regexp-opt cpsa-types 'words))
  (setq cpsa-font-lock-keywords
        `(
          (,cpsa-keywords-regexp . font-lock-keyword-face)
          (,cpsa-functions-regexp . font-lock-function-name-face)
          (,cpsa-types-regexp . font-lock-type-face)))
  (setq font-lock-defaults '((cpsa-font-lock-keywords))))

(add-to-list 'auto-mode-alist '("\\.cpsa$" . cpsa-mode))
```

Go Mode

```
(require 'go-autocomplete)

(add-hook 'go-mode-hook
          (lambda ()
            (go-eldoc-setup)
            (add-hook 'before-save-hook 'gofmt-before-save)))
```

Themes

Load solarized-light if in a graphical environment. Load the wombat theme if in a terminal.

```
(if window-system
    (load-theme 'solarized-light t)
  (load-theme 'wombat t))
```

Color Codes

Running things like RSpec in compilation mode produces ansi color codes that aren't properly dealt with by default. This takes care of that and makes sure that the colors that are trying to be presented are rendered correctly.

```
(require 'ansi-color)
(defun colorize-compilation-buffer ()
  (toggle-read-only)
  (ansi-color-apply-on-region (point-min) (point-max))
  (toggle-read-only))
(add-hook 'compilation-filter-hook 'colorize-compilation-buffer)
```

Author: Aaron Bedra

Created: 2018-06-15 Fri 08:15

Validate