Project Architecture for Devlin Ihmacs

Overview

Devlin Ihmacs is an Emacs-like text editor implemented in Python. As a MVP, it needs to be able to load text from a file in a buffer, manipulate the buffer via the keyboard, and write the edited buffer to a file.

Ihmacs runs in a terminal environment.

Interactivity

Ihmacs is controlled via the keyboard in a terminal environment. Editing functions (or methods) will be mapped to via the keymap.

Visuals and input are handled via ncurses.

Inputs will be used to construct "keychords", which is how Emacs strings together key sequences. A keychord could be `C-x C-f`, which means press Control+x and then Controll+f. This would map to the `find_file` function.

Model

Everything is tied together under an Ihmacs class. There will be a buffer class, which stores text and various information regarding how you can edit it, as well as classes for modes.

Editing functions are defined globally, these functions and take in a buffer instance as an argument. They can then run buffer methods on the passed buffer to manipulate it, as well as other functions defined globally.

Ihmacs

A class representing the global state of the editor.

Note: I am not implementing the concepts of multiple frames, nor even split windows. That would be beyond the scope of this project. There will just be one "active" buffer which is focused.

Things implemented here

- Kill ring, a list of strings representing text that has been killed (infinite clipboard baby!)
- Buffers, a list of all active buffers.
- Global keymap
- The ncurses window

Buffers

A class storing

- text: A string representing the text in the buffer
- modified: A bool representing if the buffer has been modified since last save.
- point: An int representing cursor position
- mark: An int representing the position of the mark (define region for selection for those not familiar with Emacs)
- name: A string representing buffer name.
- path: A string representing the buffer file path. If the buffer is not associated with a file, this is the empty string.
- major_mode: The major mode as a major mode object.
- *beyond MVP* minor_modes: The active minor modes as a list of minor mode objects. Minor modes would be nice but aren't needed functionality.
- keymap: Buffer keymap, derived from combining the global keymap and the modemaps.
- *beyond MVP* history: A list of commands that have been executed on the buffer. Uses a list as a stack.
- *beyond MVP* modeline: A string that is displayed in the modeline (format method applied). Shows information like current line, active major and minor modes, etc.
- display_line: An int representing where the view of the buffer starts in a window. This line is the first line in the window.

Major Mode

All major modes inherent from the "fundamental" mode class.

The major mode class will contain

- Rules for syntax highlighting (would pygments work with curses, or would I need to do some regex hell?)
- Rules for indentation
- A major mode map that is a map of keychords to editing commands.
- Mode specific (language specific) editing commands (methods).

Minor Modes (Beyond MVP)

If I have time I'll implement this.

View

Everything will be represented via ncurses in a terminal.

There will be one active buffer. This will be displayed in an ncurses window.

The window will also display a modeline at the bottom of the screen.

There will be the "echo area"/"minibuffer" on the last line. This is where new messages are displayed and where you can input text for commands that take input (say something like `M-x` (`execute_extended_command`)).

The view class will have a refresh method that is run after every keychord. It will redrew everything. Refreshing the entire screen is for simplicity; hopefully it doesn't cause obnoxious blinking for every keypress.

The view class will have an associated ncurses window.

Controller

The controller class will also be associated with the same ncurses window.

The controller class will have a method to read keychords. It will loop, pulling character input from ncurses. The input will be compared to the keymap tree, if it terminates with a function (say, you press `a` and `a` maps to `self_insert_character`) it will run the associated function on the active buffer. If it gets another dictionary (more tree) it will keep reading and either spit out and error (like `C-c n` is undefined, basically saying nothing is mapped to that), or keep going until a function is found.