Principia Softwarica: The Text Editor `efuns`

version 0.5

Yoann Padioleau
yoann.padioleau@gmail.com

with code from
Fabrice Le Fessant

March 4, 2026

Contents

Chapter 1

Introduction

This book presents the complete source code of `efuns`, a text editor written in OCaml by Fabrice Le Fessant at INRIA. Efuns is an Emacs clone—it shares Emacs's concepts of buffers, windows, modes, and keymaps—but implemented in OCaml rather than C+Lisp. The choice of OCaml means the editor can be extended simply by adding modules and recompiling, rather than needing a separate extension language.

The goal of this book is to explain with full details the source code of a text editor.

1.1 Motivations

Why a text editor? Because I think you are a better programmer if you fully understand how things work under the hood, and the editor is probably the tool a programmer uses the most.

Even if the kernel and compiler are essential to run and produce programs, and are the subjects of many textbooks, the text editor is also an essential program. Indeed, The source code of almost all programs, including kernels and compilers, was first entered in a computer by a programmer using a text editor. In fact, text editors are used to enter the code of text editors, leading to bootstrapping issues, just like in a compiler.

Here are a few questions I hope this book will answer:

- How the text of a file is represented in memory? Which data structure is used?

1.2 The text editor `efuns`

I chose `efuns` for this book because it is a clean, elegant Emacs clone written in OCaml by Fabrice Le Fessant at INRIA. It covers all the essential concepts of a screen editor — buffers, windows, modes, keymaps — while remaining small enough (under 15,000 lines) to explain completely. Using OCaml also provides an interesting contrast with the other books in the Principia Softwarica series, which focus on C programs. For a desktop application like a text editor, a higher-level language makes sense: OCaml's type system catches many bugs at compile time, pattern matching makes dispatch logic clearer, and the garbage collector removes the need for manual memory management. A key design difference from GNU Emacs is that `efuns` has no separate extension language. In Emacs, the core is written in C and extensions in Emacs Lisp, which made sense because C is harder to program in than Lisp. But with OCaml as the implementation language, you can extend the editor simply by adding a module and recompiling — no need for an embedded interpreter. This also means you can use standard OCaml debugging tools (like `ocamldebug`) rather than Emacs's ad-hoc `debug-on-error` mechanism.

1.3 Other editors

Text editors have evolved through several generations. The earliest (QED, `ed`) were line-oriented: you addressed text by line numbers and never saw more than one line at a time. The second generation—`vi`, Emacs—brought full-screen editing. A third generation (Sublime Text, VS Code, Atom) added modern UI conventions like multiple cursors and fuzzy-find palettes, while IDEs (Eclipse, IntelliJ) integrated compilation, debugging, and refactoring. Efuns sits squarely in the second generation, inheriting the Emacs model, but its codebase is small enough to study as a whole.

1.4 Other Emacs-like editors

Several Emacs clones were considered for this book. GNU Emacs itself was ruled out because of sheer size—`xdisp.c` alone is 30,000 lines, more than all of `efuns`. Microemacs and `mg` are smaller but written in C, which is already well covered elsewhere in the Principia Softwarica series. Qemacs has impressive Unicode and HTML rendering but is also C. Efuns stood out as a clean, complete Emacs clone in a higher-level language, making it both a manageable size and an interesting change of perspective.

Here are a few Emacs clones that I considered for this book, but which I ultimately discarded:

- Emacs
- Microemacs
- Qemacs
- mg

1.5 Getting started

1.6 Requirements

1.7 About this document

This document is a *literate program* [Knu92]. It derives from a set of files processed by a tool, `syncweb` [Pad09], generating either this book or the actual source code of the program. The code and its documentation are thus strongly connected.

1.8 Copyright

Most of this document is actually source code from INRIA, so those parts are copyright by INRIA. The prose is mine and is licensed under the All Rights Reserved License.

⟨*copyright header efuns* 9⟩≡ (? ? ? ? ? ? ? ? ? ? ? 0—1)

```
(***********************************************************************)
(*                                                                     *)
(*                              Efuns                                  *)
(*                                                                     *)
(*       Fabrice Le Fessant, projet Para/SOR, INRIA Rocquencourt       *)
(*                                                                     *)
(*  Copyright 1999 Institut National de Recherche en Informatique et   *)
(*  Automatique.  Distributed only by permission.                      *)
(*                                                                     *)
(***********************************************************************)
```

1.9 Acknowledgments

I would like to acknowledge of course the author of `efuns`, Fabrice Le Fessant, who wrote in some sense most of this book.

Chapter 2

Overview

2.1 Text editor principles

A screen text editor must solve several interrelated problems: represent file content in memory efficiently (the gap buffer), display that content on screen (handling tabs, control characters, and long lines), map keyboard input to editing actions (via keymaps), and keep everything synchronized—when text is inserted or deleted, all cursors, line counts, and display state must be updated consistently. The Emacs design that `efuns` follows adds further structure: "everything is a buffer" (even the minibuffer prompt and compilation output are buffers), extensibility through major and minor modes that customize keybindings and behavior per file type, and a hierarchical variable system (buffer-local, mode-local, global) for configuration.

2.2 `efuns` interfaces

Efuns is launched from the command line with optional filenames. As with Emacs, there is also a client program (`efuns_client`) that can tell a running instance to open a file, allowing integration with other tools.

2.2.1 Command-line interface

⟨*constant* `Efuns.usage_str` 11⟩≡ (? 0—1)

```
let usage_str =
 "A small editor entirely written in Objective Caml
by Fabrice LE FESSANT, INRIA Rocquencourt, FRANCE
http://pauillac.inria.fr/efuns
Options:
"
```

2.2.2 Graphical user interface

2.3 Code organization

The source is split into three layers. The *internals* provide the core data structures: `text.ml` (the gap buffer, self-contained), `efuns.ml` (the central, mutually-recursive type definitions), `ebuffer.ml`, `window.ml`, `frame.ml`, `top_window.ml`, `keymap.ml`, and `minibuffer.ml`. The *functions* layer implements editing operations on top of these structures: `simple.ml` for basic actions like insertion and deletion, `complex.ml` for operations involving the minibuffer, `search.ml`, `compile.ml`, and so on. Finally, the *major modes* (`ocaml_mode.mll`, `c_mode.mll`, `tex_mode.mll`, etc.) provide per-language syntax highlighting and indentation via `ocamllex` lexers.

2.4 Software architecture

The data structures form a layered hierarchy: `Text.t` (the gap buffer holding raw content) is wrapped in `Ebuffer.t` (which adds a name, filename, keymap, and modes), which is displayed in a `Frame.t` (a view of a buffer with its own cursor position and display state), which lives inside a `Window.t` (the geometry of a tiled subwindow), which belongs to a `Top_window.t` (the actual OS window). The global `Editor.t` holds all buffers, frames, and top windows. The control flow is event-driven: the GTK event loop dispatches key presses to `Top_window.handle_key`, which looks up the binding in the current frame's keymap (cascading through minor mode, major mode, buffer, and global keymaps), invokes the resulting action function, and then triggers a redisplay. The beauty of this design is that action functions only need to modify the buffer state (e.g., move a point or insert text)—the redisplay engine automatically handles all the visual consequences.

2.4.1 Trace of a file loading

Opening a file triggers a chain: `main` calls `Ebuffer.read_fileX` which reads the file into a `Text.t` gap buffer, wraps it in an `Ebuffer.t` with default keymap and modes, creates a `Frame.t` to display it, and finally triggers the initial redisplay.

2.4.2 Trace of a line rendering

Rendering a single line follows a pipeline: on-disk bytes are loaded into `text.text_string` (the gap buffer's backing store), line boundaries are recorded in `text.text_newlines`, then each line is expanded into boxes that handle special characters (tabs become spaces, control characters become `^X` notation), and these boxes are placed into `frame.frm_table` rows—splitting into multiple screen rows when a line overflows the window width—before finally being drawn on screen.

2.4.3 Trace of a key press

A key press flows through the GTK event loop to `Top_window.handle_keyX`, which looks up the key (or key sequence) in the current frame's keymap chain via `Keymap.try_mapX`. The matching action function is called with the current frame as argument—for example, `M->` simply moves the point to the end of the buffer. After the action returns, the event loop calls the redisplay engine, which notices the point has moved and redraws accordingly. The elegance is that action code only modifies buffer state; all visual consequences are handled automatically by redisplay.

2.5 Book structure

The book follows a layered approach. It begins with the core data structures (gap buffer, types, buffers, frames, windows), then walks through `main` top-down. Next it covers the editing functions bottom-up from simple to complex, followed by the display engine and the major modes. The appendices contain auxiliary code such as X11 bindings and configuration.

Chapter 3

Core Data Structures

The core types are mutually recursive (defined together in `Efuns`) and form a containment hierarchy: `Text.t` ⊂ `Ebuffer.t` ⊂ `Frame.t` ⊂ `Window.t` ⊂ `Top_window.t` ⊂ `Editor.t`. The control structures—`Action.t` (named functions), `Keymap.t` (key-to-action mappings), and `Var.t` (typed configuration variables)—cut across this hierarchy, connecting keyboard input to buffer modifications.

3.1 `Text.t`

`Text.t` is the lowest-level data structure: a gap buffer holding the raw file content as a mutable byte array. The gap (a hole in the array) sits at the current editing position, making sequential insertions O(1). The `text_newlines` array maintains an index of line boundaries for fast line-number lookups. Two coordinate systems coexist: user positions (what the editor user sees, ignoring the gap) and internal positions (offsets into `text_string` including the gap). The `Text.get_position`/`Text.set_position` functions convert between them.

⟨*type* `Text.t` 13a⟩≡ (? 0—1)

```
type t = {
    mutable text_string : bytes;
    mutable text_size : int; (* Bytes.length text.text_string *)

    mutable text_newlines : line array;
    mutable text_nlines : int; (* Array.length text.text_newlines *)

    ⟨Text.t gap fields 37a⟩
    ⟨Text.t history fields 14a⟩
    ⟨Text.t attribute fields 47c⟩
    ⟨Text.t other fields 39b⟩
  }
```

⟨*type* `Text.position` 13b⟩≡ (? 0—1)

```
type position = int
```

⟨*type* `Text.line` 13c⟩≡ (? 0—1)

```
type line = {
    mutable position : position2; (* bol (beginning of line) *)

    ⟨Text.line representation fields 52f⟩
    ⟨Text.line other fields 54b⟩
  }
```

⟨*type* `Text.version` 13d⟩≡ (? 0—1)

```
type version = int
```

⟨Text.t *history fields* 14a⟩≡ (13a) 107b▷
```
mutable text_modified : version;
```

⟨*function* Text.version 14b⟩≡ (? 0—1)
```
let version text =
  text.text_modified
```

⟨*function* Text.create 14c⟩≡ (? 0—1)
```
let create str =
  let newlines = compute_newlines str in
  let attrs = (Array.make (String.length str) direct_attr) in

    {
      text_string = (Bytes.of_string str);
      text_size = String.length str;

      text_newlines = newlines;
      text_nlines = Array.length newlines;

      text_attrs = attrs;

      (* no gap at the beginning *)
      gpoint = { pos = 0; line = 0 };
      gsize = 0;
      text_points = [];

      text_modified = 0;
      text_clean = false;
      text_history = [];
      text_readonly = false;
    }
```

⟨*function* Text.compute_newlines 14d⟩≡ (? 0—1)
```
let compute_newlines string =
  let (nbr_newlines,_) = Utils.count_char string '\n' in
  let newlines = Array.make (nbr_newlines + 2) (mk_line_with_pos 0) in
  let curs = ref 0 in
  (* newlines.(0) is already set with 0 position for its bol *)
  for i = 1 to nbr_newlines do
    let pos = String.index_from string !curs '\n' in
    newlines.(i) <- mk_line_with_pos (pos+1);
    curs := pos + 1;
  done;
  newlines.(nbr_newlines+1) <- mk_line_with_pos (String.length string + 1);
  newlines
```

⟨*function* Text.nbr_lines 14e⟩≡ (? 0—1)
```
let nbr_lines text =
  text.text_nlines - 2
```

⟨*function* Text.mk_line_with_pos 14f⟩≡ (? 0—1)
```
let mk_line_with_pos pos =
  {
   position = pos;

   boxes = [];
   repr_string = Bytes.of_string "";
   repr_len = 0;
   line_modified = true;
  }
```

3.2 Ebuffer.t

A buffer wraps a `Text.t` with editor-level metadata: a unique name, an optional filename, a major mode, minor modes, a local keymap, and a syntax table. It also holds `buf_point` and `buf_start`—saved cursor and viewport positions that are restored when switching back to this buffer from another one. The buffer's `buf_name` serves as a primary key in the editor's buffer hash table, and is made unique by appending a counter suffix (e.g., `file.txt<2>`) when needed.

⟨*type* `Efuns.buffer` 15a⟩≡ (? 0—1)
```
(* a buffer containing a file content in buffer.buf_text *)
and buffer =
  {
    mutable buf_text : Text.t;

    (* should be unique; can be used as a primary key in editor.edt_buffers *)
    mutable buf_name : string;
    mutable buf_filename : string option;

    ⟨Efuns.buffer position fields 15d⟩
    ⟨Efuns.buffer history fields 15e⟩
    ⟨Efuns.buffer other fields 18a⟩
  }
```
Uses `Efuns.map` and `Text.point`.

⟨*type* `Ebuffer.t` 15b⟩≡ (? 0—1)
```
type t = Efuns.buffer
```

⟨*type* `Text.point` 15c⟩≡ (? 0—1)
```
type point = {
    mutable pos : position2;
    mutable line : int;
}
```

⟨`Efuns.buffer` *position fields* 15d⟩≡ (15a)
```
mutable buf_point : Text.point;
mutable buf_start : Text.point;
```

⟨`Efuns.buffer` *history fields* 15e⟩≡ (15a) 59d▷
```
mutable buf_modified : int; (* version *)
```

⟨*function* `Ebuffer.create` 15f⟩≡ (? 0—1)
```
let create name filename text local_map =
  let name = get_unique_name name in
  let buf =
    {
      buf_text = text;

      buf_name = name;
      buf_filename = filename;

      buf_point = Text.new_point text;
      buf_start = Text.new_point text;

      buf_last_saved = Text.version text;
      buf_modified = 0;

      buf_map = local_map;

      buf_syntax_table = default_syntax_table;
```

```
      buf_map_partial = true;
      buf_vars = Store.new_store ();
      buf_major_mode = fondamental__mode;
      buf_minor_modes = [];

      buf_sync = false;
      buf_mark = None;
      buf_shared = 0;
      buf_finalizers = [];
      buf_history_pos = [||];

      ⟨Ebuffer.create() buffer other fields setup 53d⟩

    } in
  ⟨Ebuffer.create() adjust editor global fields 20d⟩
  ⟨Ebuffer.create() adjust charreprs 53e⟩
  ⟨Ebuffer.create() run hooks 28c⟩
  buf
```

3.3 Frame.t

A frame is a view of a buffer. It holds its own cursor (`frm_point`), viewport start (`frm_start`), and display table (`frm_table`) mapping screen lines to text lines. Multiple frames can show the same buffer simultaneously. The frame also tracks display state—version numbers for incremental redraw, a `frm_cutline` for line wrapping, and offsets for scrolling within wrapped lines. The terminology can be confusing: what `efuns` calls a "frame" corresponds to an Emacs "window" (a pane showing a buffer), while what `efuns` calls a "window" is purely geometry (position and dimensions within the top window). An Emacs "frame" (a separate OS window) corresponds to `efuns`'s `Top_window.t`.

⟨*type* Efuns.frame 16a⟩≡ (? 0—1)

```
(* a frame: a view of a buffer for a window *)
and frame  =
  {
    mutable frm_buffer : buffer;

    mutable frm_xpos : int;
    mutable frm_ypos : int;

    mutable frm_width : int;
    mutable frm_height : int; (* including status line *)

    ⟨Efuns.frame current position fields 16c⟩
    ⟨Efuns.frame window fields 18d⟩
    ⟨Efuns.frame decoration fields 64b⟩
    ⟨Efuns.frame status field 74a⟩
    ⟨Efuns.frame history fields 65b⟩
    ⟨Efuns.frame other fields 65d⟩
    caps: < frame_caps >;
  }
```

Uses Efuns.window 19b and Text.point.

⟨*type* Frame.t 16b⟩≡ (? 0—1)

```
type t = Efuns.frame
```

⟨Efuns.frame *current position fields* 16c⟩≡ (16a) 64e▷

```
(* insert point *)
mutable frm_point : Text.point;
```

Uses Efuns.status 81b.

⟨*function* `Frame.create` 17a⟩≡ (? 0—1)

```
let create (caps : < frame_caps; .. >) (window : Window.t) (mini : string option) (buf : Ebuffer.t) : t =
  let frame = create_without_top caps window mini buf in
  ⟨Frame.create() adjust active frame 19f⟩
  frame
```

⟨*function* `Frame.create_without_top` 17b⟩≡ (? 0—1)

```
let create_without_top (caps : < frame_caps; ..>) (window : Window.t) (mini : string option) (buf : Ebuffer.t)
  let text = buf.buf_text in

  let frm_start = Text.dup_point text buf.buf_start in
  let point     = Text.dup_point text buf.buf_point in
  let frm_end   = Text.dup_point text buf.buf_start in (* ?? *)

  buf.buf_shared <- buf.buf_shared + 1;

  ⟨Frame.create_without_top() let status 74c⟩

  let frame =
    { frm_buffer = buf;

      frm_window = window;
      frm_xpos = window.win_xpos;
      frm_ypos = window.win_ypos;
      frm_width = window.win_width;
      frm_height = window.win_height;

      frm_start = frm_start;
      frm_point = point;
      frm_end = frm_end;

      (* this will be set later in set_cursor called from update_display *)
      frm_cursor_x = 0;
      frm_cursor_y = 0;
      frm_cursor = Bytes.make 1 ' ';
      frm_cursor_attr = Text.direct_attr;

      frm_last_text_updated = 0;
      frm_last_buf_updated = 0;

      frm_redraw = true; (* ! *)
      frm_cutline = window.win_width - 1; (* ! *)

      frm_force_start = false;
      frm_x_offset = 0;
      frm_y_offset = 0;
      frm_table = [||];

      frm_has_status_line = 1;
      frm_status = status;
      frm_mini_buffer = mini;

      frm_prefix = [];
      frm_last_action = Keymap.dummy_action;

      frm_killed = false;
      caps = (caps :> < frame_caps >);
    }
  in
  ⟨Frame.create_without_top() adjust status of frame 74f⟩
```

```
    install window frame;
    frame
```

⟨Efuns.buffer *other fields* 18a⟩≡ (15a) 23b ▷
```
  mutable buf_shared : int; (* number of frames for that buffer *)
```

3.4 Window.t

A window is just geometry: position and dimensions in character units. The interesting part is `win_down`, which forms a binary tree: a window is either a leaf containing a `Frame.t`, or a horizontal or vertical combination (`HComb`/`VComb`) of two sub-windows. The `win_up` link points back to the parent window or the top window, enabling navigation from any frame to its containing top window via `Window.top`.

⟨*type* Efuns.window 18b⟩≡ (? 0—1)
```
  and window =
    {
      mutable win_xpos : int;
      mutable win_ypos : int;

      mutable win_width : int;
      mutable win_height : int; (* including status line *)

      ⟨Efuns.window other fields 18e⟩
    }
```
Uses Efuns.frame, Efuns.top_window 66d, and Efuns.window 19b.

⟨*type* Window.t 18c⟩≡ (? 0—1)
```
  type t = Efuns.window
```

⟨Efuns.frame *window fields* 18d⟩≡ (16a)
```
  mutable frm_window : window;
```

⟨Efuns.window *other fields* 18e⟩≡ (18b) 78a ▷
```
  mutable win_down : window_down;
  mutable win_up : window_up;
```
Uses Efuns.window 19b.

⟨*type* Efuns.window_down 18f⟩≡ (? 0—1)
```
  and window_down =
  | WFrame of frame
  | NoFrame

  | HComb of window * window
  | VComb of window * window
```
Uses Efuns.top_window 66d.

⟨*type* Efuns.window_up 18g⟩≡ (? 0—1)
```
  and window_up =
    Window of window
  | TopWindow of top_window
```

⟨*function* `Window.create` 19a⟩≡ (? 0—1)

```
let create mini up_window xpos ypos width height =
  {
  win_xpos = xpos;
  win_ypos = ypos;

  win_width = width;
  win_height = height;

  win_down = NoFrame;
  win_up = up_window;

  win_mini = mini;
  }
```

3.5 `Top_window.t`

A `Top_window.t` represents an actual OS-level window (historically an X11 widget). It sits at the top of the containment hierarchy: it owns the root `Window.t` (which recursively contains all the frames), tracks overall dimensions, holds the graphics backend, and knows which frame is currently active (`top_active_frame`). It is kept separate from `Editor.t` to allow—in principle—multiple top-level windows sharing the same editor state.

⟨*type* `Efuns.top_window` 19b⟩≡ (? 0—1)

```
(* a window containing some frames *)
and top_window =
  {
    mutable top_width : int;
    mutable top_height : int; (* including minibuffer line *)

    mutable window : window;

    ⟨Efuns.top_window other fields 19d⟩
  }
```

⟨*type* `Top_window.t` 19c⟩≡ (? 0—1)

```
type t = Efuns.top_window
```

⟨`Efuns.top_window` *other fields* 19d⟩≡ (19b) 19e▷

```
mutable graphics : Xdraw.graphics_backend option;
```

⟨`Efuns.top_window` *other fields* 19e⟩+≡ (19b) ◁19d 73d▷

```
mutable top_active_frame : frame;
```

⟨`Frame.create()` *adjust active frame* 19f⟩≡ (17a)

```
let top_window = Window.top window in
top_window.top_active_frame <- frame;
```

⟨*function* `Window.top` 19g⟩≡ (? 0—1)

```
let rec top window =
  match window.win_up with
  | TopWindow top_window -> top_window
  (* recurse *)
  | Window window -> top window
```

3.6 Editor.t

The editor is the global "world" object, holding hash tables of all buffers (keyed by name and by filename), the list of top windows, and global display settings. It is stored as a global mutable reference (`global_editor`) rather than being threaded through every function call. Originally called "location" (reflecting `efuns`'s Jocaml ancestry), it was renamed to the more intuitive `editor`.

⟨*type* Efuns.editor 20a⟩≡ (? 0—1)
```
type editor =
  {
    (* key is buffer name (made unique via get_unique_name()) *)
    mutable edt_buffers : (string, buffer) Hashtbl.t;
    (* key is filename (should be unique? todo: use realpath?) *)
    mutable edt_files : (string (* filename *), buffer) Hashtbl.t;

    (* list when have one efuns running multiple top windows *)
    mutable top_windows : top_window list;

    (* pwd of efuns when started and then dirname of file of active frame *)
    mutable edt_dirname : string;

    (* general look, configurable via -xxx command line options or .efunsrc
     * (dimension type is in characters)
     *)
    mutable edt_width : int;
    mutable edt_height : int;

    mutable edt_fg : string;
    mutable edt_bg : string;
    mutable edt_font : string;

    ⟨Efuns.editor other fields 23c⟩
  }
```
Uses `Efuns.map` and `Store.t`.

⟨*global* Efuns.global_editor 20b⟩≡ (? 0—1)
```
let global_editor = ref None
```

⟨*function* Efuns.editor 20c⟩≡ (? 0—1)
```
let editor () =
  match !global_editor with
  | None -> failwith "no global editor defined"
  | Some x -> x
```

⟨Ebuffer.create() *adjust editor global fields* 20d⟩≡ (15f)
```
Hashtbl.add (Globals.editor()).edt_buffers name buf;
```

⟨*function* Frame.find_buffer_frame 20e⟩≡ (? 0—1)
```
let find_buffer_frame buf =
  try
    (Globals.editor()).top_windows |> List.iter (fun top_window ->
      top_window.window |> Window.iter (fun frame ->
        if Eq.phys_equal frame.frm_buffer buf
        then raise (FoundFrame frame)
      )
    );
    raise Not_found
  with FoundFrame frame -> frame
```

⟨*exception* `Frame.FoundFrame` 21a⟩≡ (? 0—1)
```
exception FoundFrame of frame
```

3.7 `Action.t`

An action is simply a function `frame -> unit`. Every editing operation—inserting a character, moving the cursor, opening a file—is an action. Actions take the active frame as their sole parameter and can reach the buffer, text, window, and editor from there. They are registered by name in a global hash table (`Efuns.actions`) so they can be invoked by name from `M-x`, menus, and configuration files.

⟨*type* `Efuns.action` 21b⟩≡ (? 0—1)
```
and action = frame -> unit
```

⟨*type* `Action.t` 21c⟩≡ (? 0—1)
```
type t = Efuns.action
```

⟨*type* `Efuns.generic_action` 21d⟩≡ (? 0—1)
```
and generic_action =
| FrameAction of action
| BufferAction of (buffer -> unit)
```
Uses `Efuns.action` and `Efuns.map`.

⟨*function* `Keymap.dummy_action` 21e⟩≡ (? 0—1)
```
let dummy_action _frame = ()
```

3.7.1 Action name

Every editing command is registered by name in a global `actions` hash table, mapping strings like `"forward_char"` to their `generic_action` function. This is the mechanism behind `M-x`: the user types an action name and efuns looks it up in the table. It also enables saving keybindings in configuration files as strings rather than function pointers. Since OCaml lacks `eval`, `define_actionX` must be called explicitly for each command.

⟨*type* `Efuns.action_name` 21f⟩≡ (? 0—1)
```
and action_name = string
```
Uses `Efuns.key`.

⟨*global* `Efuns.actions` 21g⟩≡ (? 0—1)
```
let (actions : (action_name, generic_action) Hashtbl.t) =
  Hashtbl.create 63
```

⟨*function* `Efuns.define_action` 21h⟩≡ (? 0—1)
```
let define_action action_name action_fun =
  ⟨sanity check action defined twice 21j⟩
  Hashtbl.add actions action_name (FrameAction action_fun)
```

⟨*function* `Efuns.define_buffer_action` 21i⟩≡ (? 0—1)
```
let define_buffer_action action_name action_fun =
  ⟨sanity check action defined twice 21j⟩
  Hashtbl.add actions action_name (BufferAction action_fun)
```

⟨*sanity check action defined twice* 21j⟩≡ (21)
```
(try
    Hashtbl.find actions action_name |> ignore;
    Error.error (spf "action \"%s\" defined twice" action_name);
 with _ -> ()
);
```

⟨*function* `Efuns.get_action` 22a⟩≡ (? 0—1)

```
let get_action action_name =
  try Hashtbl.find actions action_name
  with Not_found ->
    Error.error_exn (spf "Could not find action %s. Forgot define_action()?"
        action_name) Not_found;
    BufferAction (fun _ -> ())
```

⟨*function* `Efuns.execute_action` 22b⟩≡ (? 0—1)

```
let execute_action action_name =
  match (get_action action_name) with
  | FrameAction f -> f
  | BufferAction f -> (fun frame -> f frame.frm_buffer)
```

3.8 `Keymap.t`

A keymap maps key combinations to actions. It uses a dual structure for performance: `char_map` is a 256-entry array for single-character bindings (O(1) lookup), while `complex_bindings` is a list for modified keys and multi-key sequences. Bindings cascade through four scopes: minor mode → major mode → buffer-local → global, so a mode like shell-mode can override `Return` without affecting other buffers. Prefix keys (like `C-x`) bind to sub-keymaps, enabling arbitrarily deep key sequences.

⟨*type* `Efuns.map` 22c⟩≡ (? 0—1)

```
type map =
  {
    (* 256 array, one character simple key = one action *)
    char_map : binding array;
    (* complex key, possible sub maps *)
    mutable complex_bindings : (key * binding) list;
  }
```

⟨*type* `Keymap.t` 22d⟩≡ (? 0—1)

```
type t = Efuns.map
```

⟨*type* `Efuns.key` 22e⟩≡ (? 0—1)

```
and key = mod_ident * keySym
```

⟨*type* `Efuns.mod_ident` 22f⟩≡ (? 0—1)

```
and mod_ident =
  NormalMap
| ControlMap
| MetaMap
| ControlMetaMap
```

⟨*type* `Efuns.keySym` 22g⟩≡ (? 0—1)

```
and keySym = int
```

⟨*type* `Efuns.keys` 22h⟩≡ (? 0—1)

```
and keys = key list
```

Uses `Text.t`.

⟨*type* `Efuns.binding` 22i⟩≡ (? 0—1)

```
and binding =
  Function of action
| Prefix of map
| Unbound
```

Uses `Text.point`.

⟨*function* `Keymap.create` 23a⟩≡ (? 0—1)
```
let create () =
  { char_map = Array.make 256 Unbound;
    complex_bindings = [];
  }
```

⟨`Efuns.buffer` *other fields* 23b⟩+≡ (15a) ◁18a 23e▷
```
buf_map : map;
```

⟨`Efuns.editor` *other fields* 23c⟩≡ (20a) 26c▷
```
edt_map : map;
```

⟨*function* `Ebuffer.get_binding` 23d⟩≡ (? 0—1)
```
let get_binding buf keylist =
  let binding = ref Unbound in
  try
    ⟨Ebuffer.get_binding() minor mode key search 133a⟩
    ⟨Ebuffer.get_binding() major mode key search 129b⟩
    (let b = Keymap.get_binding buf.buf_map keylist in
      match b with
      | Prefix _map -> binding := b;
      | Function _f -> binding := b; raise Exit
      | Unbound -> ()
    );
    ⟨Ebuffer.get_binding() if partial map 23f⟩
    !binding
  with Exit -> !binding
```

⟨`Efuns.buffer` *other fields* 23e⟩+≡ (15a) ◁23b 26b▷
```
mutable buf_map_partial : bool;
```

⟨`Ebuffer.get_binding()` *if partial map* 23f⟩≡ (23d)
```
if buf.buf_map_partial then
  (let b = Keymap.get_binding (Globals.editor()).edt_map keylist in
    match b with
    | Prefix _map -> binding := b;
    | Function _f -> binding := b; raise Exit
    | Unbound -> ()
  );
```

⟨*function* `Keymap.add_global_key` 23g⟩≡ (? 0—1)
```
let add_global_key prefix action =
  add_binding (Globals.editor()).edt_map prefix action
```

⟨*function* `Keymap.add_local_key` 23h⟩≡ (? 0—1)
```
let add_local_key buf prefix action =
  add_binding buf.buf_map prefix action
```

3.8.1 Configurable keymap

The `global_map` option allows users to override the default keymap entirely from their `.efunsrc` configuration file. Each entry is a pair of key sequence and action name (as a string). If the user has not customized it (the list is empty), `Config.init_global_mapX` falls back to the built-in `core_map` and `standard_map`.

⟨*constant* `Config.global_map` 23i⟩≡ (? 0—1)
```
let global_map = define_option ["global_map"] ""
  (list_option Keymap.binding_option) []
```

⟨*constant* `Simple.binding_option` 24a⟩≡ (? 0—1)
```
let binding_option =
  Options.tuple2_option (Options.smalllist_option key_option,
                         Options.string_option)
```

⟨*constant* `Simple.key_option` 24b⟩≡ (? 0—1)
```
let key_option = Options.define_type "Key" value_to_key key_to_value
```

⟨*function* `Config.init_global_map` 24c⟩≡ (? 0—1)
```
let init_global_map () =
  if !!global_map =*= []
  then
    (core_map @ standard_map) |> List.iter (fun (keys, action) ->
        Keymap.add_global_key keys action
    )
  else
  !!global_map |> List.iter (fun (keys, action) ->
      try
        Keymap.add_global_key keys (Action.execute_action action)
      with e ->
        Log.printf "Error for action %s" action;
        Log.exn "%s\n" e;
  );

  (* Mouse *)
  ⟨Config.init_global_map() mouse keys setup 85d⟩
  ()
```
Uses `Default_config.core_map`, `Default_config.global_map`, `Default_config.standard_map`, `Efuns.mod_ident.NormalMap`, and `XK.xk_Pointer_Button1`.

3.8.2 Standard Emacs/Efuns keymap

The `standard_map` defines the familiar Emacs keybindings: `C-f`/`C-b` for character movement, `C-x C-f` for opening files, `C-w` for kill-region, and so on. Three prefix keys (`C-x`, `C-c`, `C-h`) are given named constants because they appear in many multi-key sequences. The bindings are organized by category (loading, navigating, editing, searching, etc.) and each entry is a pair of key sequence and action function—not a string name, since this is the compiled-in default rather than a user configuration file.

⟨*constant* `Keymap.c_x` 24d⟩≡ (? 0—1)
```
let c_x = (ControlMap, Char.code 'x')
```

⟨*constant* `Keymap.c_c` 24e⟩≡ (? 0—1)
```
let c_c = (ControlMap, Char.code 'c')
```

⟨*constant* `Keymap.c_h` 24f⟩≡ (? 0—1)
```
let c_h = (ControlMap, Char.code 'h')
```

⟨*constant* `standard_map` 24g⟩≡ (? 0—1)
```
let standard_map = [
  ⟨standard_map entries 25⟩
]
```

⟨standard_map *entries* 25⟩≡ (24g)

```
(* ------------------------------------------------------- *)
(* Loading *)
(* ------------------------------------------------------- *)
⟨loading keys 58a⟩

(* ------------------------------------------------------- *)
(* Navigating (in the file) *)
(* ------------------------------------------------------- *)
⟨navigating keys 94a⟩

(* ------------------------------------------------------- *)
(* Editing *)
(* ------------------------------------------------------- *)

(* ------------- *)
(* Inserting *)
(* ------------- *)
(* see also the many self_insert_xxx in core_map *)
⟨inserting keys 101a⟩
(* ------------- *)
(* Deleting *)
(* ------------- *)
⟨deleting keys 101g⟩
(* ----------------------------- *)
(* Moving (Cut, copy, paste) *)
(* ----------------------------- *)
⟨moving keys 103a⟩
(* ---------------------- *)
(* Transforming *)
(* ---------------------- *)
⟨transforming keys 147a⟩

(* ------------------------------------------------------- *)
(* Search/replace *)
(* ------------------------------------------------------- *)
⟨searching keys 109a⟩
⟨replacing keys 114c⟩

(* ------------------------------------------------------- *)
(* Undoing *)
(* ------------------------------------------------------- *)
⟨undoing keys 106c⟩

(* ------------------------------------------------------- *)
(* External commands *)
(* ------------------------------------------------------- *)
⟨external commands keys 134a⟩

(* ------------------------------------------------------- *)
(* Buffers/windows/frames *)
(* ------------------------------------------------------- *)
⟨buffer management keys 119a⟩
⟨buffer navigating keys 121b⟩
⟨frame management keys 122d⟩
⟨frame navigation keys 124d⟩

(* ------------------------------------------------------- *)
(* Meta *)
(* ------------------------------------------------------- *)
```

⟨*meta keys* 89b⟩

```
(* ------------------------------------------------------- *)
(* Saving *)
(* ------------------------------------------------------- *)
```
⟨*saving keys* 59b⟩

```
(* ------------------------------------------------------- *)
(* Rectangles *)
(* ------------------------------------------------------- *)
[c_x; NormalMap, Char.code 'r'; NormalMap, Char.code 't'],
  Rectangle.rectangle_insert;
[c_x; NormalMap, Char.code 'r'; NormalMap, Char.code 'k'],
  Rectangle.rectangle_kill;

(* ------------------------------------------------------- *)
(* Compile/grep *)
(* ------------------------------------------------------- *)

(* ------------------------------------------------------- *)
(* Macros *)
(* ------------------------------------------------------- *)
[c_x; NormalMap, Char.code '('], Macros.start_macro;
[c_x; NormalMap, Char.code ')'], Macros.end_macro;
[c_x; NormalMap, Char.code 'e'], Macros.call_macro;

(* ------------------------------------------------------- *)
(* Misc *)
(* ------------------------------------------------------- *)
```
⟨*misc keys* 27d⟩

Uses `Efuns.mod_ident.ControlMap`, `Efuns.mod_ident.MetaMap`, `Efuns.mod_ident.NormalMap`, `XK.xk_BackSpace`, `XK.xk_Left`, and `XK.xk_Return`.

3.9 `Var.t`

Variables provide Emacs-style configuration: a typed key-value store with scoped lookup. `Var.get_var` searches buffer-local → major mode → minor modes → global, so a Python buffer can have `indent_level = 4` while a Makefile buffer has `indent_level = 8`, and both fall back to the global default if unset. The `Store` module (see appendix) uses an unsafe but clever trick with `Obj.magic` to provide type-safe heterogeneous storage without requiring all variable types to be known in advance.

⟨*type* `Var.t` 26a⟩≡ (? 0—1)
```
type 'a t = 'a Store.var
```

⟨`Efuns.buffer` *other fields* 26b⟩+≡ (15a) ◁23e 53b▷
```
mutable buf_vars : Store.t;
```

⟨`Efuns.editor` *other fields* 26c⟩+≡ (20a) ◁23c 49a▷
```
edt_vars : Store.t;
```

⟨*function* `Var.set_global` 26d⟩≡ (? 0—1)
```
let set_global var value =
  Store.set (Globals.editor()).edt_vars var value
```

⟨*function* `Var.set_local` 26e⟩≡ (? 0—1)
```
let set_local buf var value =
  Store.set buf.buf_vars var value
```

⟨*function* `Var.get_global` 27a⟩≡ (? 0—1)
```
let get_global var =
  Store.get (editor()).edt_vars var
```

⟨*function* `Var.get_local` 27b⟩≡ (? 0—1)
```
let get_local buf var =
  Store.get buf.buf_vars var
```

⟨*function* `Var.get_var` 27c⟩≡ (? 0—1)
```
let get_var buf var =
  try
    Store.get buf.buf_vars var
  with Not_found ->
    try
      ⟨Var.get_var() try with major mode variables 129c⟩
    with Not_found ->
      try
        ⟨Var.get_var() try with minor mode variables 133b⟩
      with Not_found ->
        Store.get (editor()).edt_vars var
```

⟨*misc keys* 27d⟩≡ (25) 38a▷
```
[c_h; NormalMap, Char.code 'v'], Interactive.describe_variable;
```
Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Complex.get_variable` 27e⟩≡ (? 0—1)
```
let describe_variable frame =
  Select.select frame "get_variable : " variable_hist ""
    (all_variables frame)
    (fun s -> s)
    (fun variable ->
      Top_window.mini_message frame
        (Printf.sprintf "%s : %s" variable (
          let buf = frame.frm_buffer in
          try
            Store.print buf.buf_vars variable
          with _ ->
            Store.print (Globals.editor()).edt_vars variable)))
[@@interactive]
```

⟨*constant* `Complex.variable_hist` 27f⟩≡ (? 0—1)
```
let variable_hist = ref []
```

3.10 `Option.t`

3.11 Hooks

3.11.1 Variable hooks

⟨*function* `Efuns.exec_hooks` 27g⟩≡ (? 0—1)
```
let exec_hooks hooks arg =
  hooks |> List.iter (fun f ->
    try f arg
    with exn ->
     (* less: could be nice to give names to hooks *)
     Error.error_exn  "exn in hook" exn
  )
```

⟨*function* `Efuns.add_hook` 28a⟩≡ (? 0—1)
```
let add_hook hook_var hook =
  let tail = try Var.get_global hook_var with _ -> [] in
  Var.set_global hook_var (hook :: tail)
```

⟨*constant* `Ebuffer.create_buf_hook` 28b⟩≡ (? 0—1)
```
let create_buf_hook = Store.create_abstr "create_buf_hook"
```

⟨`Ebuffer.create()` *run hooks* 28c⟩≡ (15f)
```
let hooks = try Var.get_global create_buf_hook with Not_found -> [] in
Hooks.exec_hooks hooks buf;
```

3.11.2 Named action hooks

⟨*function* `Frame.exec_named_hooks` 28d⟩≡ (? 0—1)
```
let exec_named_hooks hooks frame =
  hooks |> List.rev |> List.iter (fun action_name ->
   try Action.execute_action action_name frame with _ -> ()
  )
```

⟨*constant* `Frame.change_buffer_hooks` 28e⟩≡ (? 0—1)
```
let change_buffer_hooks = define_option ["change_buffer_hooks"] ""
    (list_option string_option)
  [ "check_file" ]
```

⟨*function* `Ebuffer.exec_named_buf_hooks` 28f⟩≡ (? 0—1)
```
let exec_named_buf_hooks hooks frame =
  hooks |> List.rev |> List.iter (fun action_name ->
    try Action.execute_buffer_action action_name frame
    with exn -> Error.error_exn "exec_named_buf_hooks" exn
  )
```

⟨*function* `Ebuffer.exec_named_buf_hooks_with_abort` 28g⟩≡ (? 0—1)
```
let exec_named_buf_hooks_with_abort hooks frame =
  hooks |> List.rev |> List.iter (fun action_name ->
    Action.execute_buffer_action action_name frame
 )
```

⟨*constant* `Ebuffer.save_buffer_hooks` 28h⟩≡ (? 0—1)
```
let save_buffer_hooks = define_option ["save_buffer_hooks"] ""
    (list_option string_option)
  [ ]
```

3.11.3 Startup hooks

Chapter 4

`main()`

The initialization sequence proceeds in a specific order: (1) parse command-line arguments, (2) load the `.efunsrc` configuration file (which sets Options), (3) create the editor instance with all its hash tables and stores, (4) run startup hooks (which register major modes, set up keybindings, etc.), (5) create the graphical top window, and (6) enter the event loop. The order matters: startup hooks must run after the editor exists but before the GUI, so modes can register themselves in time to be applied to the initial files.

⟨*toplevel* `Main` *call* `main()` 29a⟩≡ (? 0—1)

```
let _ =
  (*old: UCommon.main_boilerplate (fun () -> *)
  Cap.main (fun (caps : Cap.all_caps) ->
    (* let r = Gc.get () in r.Gc.verbose <- true; Gc.set r; *)
    let argv = CapSys.argv caps in
    Exit.exit caps
      (Exit.catch (fun () ->
         CLI.main caps argv))
  )
```

⟨*constant* `Efuns.init_files` 29b⟩≡ (? 0—1)

```
let initial_files = ref []
```

⟨*function* `Main.main` 29c⟩≡ (? 0—1)

```
let main (caps : < caps; .. >) (argv : string array) : Exit.t =
  ⟨main() set signal handlers 152e⟩

  let level = ref (Some Logs.Warning) in

  let options = [
    ⟨main() command line options 31a⟩
   ]
   in
   (try
     Arg.parse_argv argv options
       (fun name -> initial_files := name :: !initial_files)
     usage_str;
    with
    | Arg.Bad msg -> Console.eprint caps msg; raise (Exit.ExitCode 2)
    | Arg.Help msg -> Console.print caps msg; raise (Exit.ExitCode 0)
  );
  Logs_.setup !level ();
  Logs.info (fun m -> m "starting logging");

  ⟨main() set options 30a⟩

  ⟨main() initial editor 31f⟩
```

⟨main() *initialize the world* 32a⟩
⟨main() *misc initializations* 49c⟩

⟨main() *run the UI* 32f⟩
Exit.OK

4.1 Configuration file: `.efunsrc`

Efuns persists settings in a `.efunsrc` file, searched first along `load_path` then falling back to the home directory. The options (width, height, colors, font) are declared at the toplevel with `define_option` and sensible defaults, so `!!` returns usable values even before the file is loaded. The `=:=` operator writes back into an option, used below to let command-line flags override what `.efunsrc` set.

⟨main() *set options* 30a⟩≡ (29c)
⟨main() *set options filename* 30b⟩
⟨main() *adjust options* 31b⟩

⟨main() *set options filename* 30b⟩≡ (30a)
```
Options.filename :=
  (try Utils.find_in_path (Utils.homedir :: !!Globals.load_path) ".efunsrc"
   with _ -> Filename.concat Utils.homedir ".efunsrc"
  );
(try Options.load ()
 with exn -> Error.error_exn "init error" exn
);
```

⟨*function* Config.save_options 30c⟩≡ (? 0—1)
```
let save_options _frame =
  Options.save ()
[@@interactive]
```

⟨*constants Main options* 30d⟩≡ (? 0—1)
```
(* old: I used to enforce 80, but many people sometimes write in col 80
 * so simpler go a little bit above
 *)
let width      = define_option ["width"] "" int_option 82
let height     = define_option ["height"] "" int_option 44
let foreground = define_option ["foreground"] "" string_option "wheat"
let background = define_option ["background"] "" string_option "DarkSlateGray"
```

⟨*constant* Efuns.font 30e⟩≡ (? 0—1)
```
let font = define_option ["font"] "" string_option "Monospace 20"
```

4.2 Arguments processing

Command-line flags like `-width` or `-fg` cannot be written directly into the Options because `.efunsrc` has not been loaded yet at argument-parsing time. Loading the file afterwards would silently overwrite whatever the user passed on the command line. The workaround is a set of intermediate `ref None` variables: arguments are captured there first, the configuration file is loaded, and then the `adjust options` chunk applies any `Some` values with `=:=`, giving command-line flags the final say.

⟨*constants* Main.xxx_opt 30f⟩≡ (? 0—1)
```
let width_opt   = ref None
let height_opt  = ref None
let fg_opt      = ref None
let bg_opt      = ref None
let font_opt    = ref None
```

⟨main() *command line options* 31a⟩≡ (29c) 31e▷
```
  "-width"  , Arg.Int (fun i -> width_opt := Some i), "<len>: Width in chars";
  "-height" , Arg.Int (fun i -> height_opt := Some i), "<len>: Height in chars";
  "-fg"     , Arg.String(fun s -> fg_opt := Some s), "<color>: Foreground color";
  "-bg"     , Arg.String(fun s -> bg_opt := Some s), "<color>: Background color";
  "-font"   , Arg.String(fun s -> font_opt := Some s), "<font>: Font name";
```

⟨main() *adjust options* 31b⟩≡ (30a)
```
  (match !width_opt  with None -> () | Some color -> width =:= color);
  (match !height_opt with None -> () | Some color -> height =:= color);
  (match !fg_opt     with None -> () | Some color -> foreground =:= color);
  (match !bg_opt     with None -> () | Some color -> background =:= color);
  (match !font_opt   with None -> () | Some color -> Globals.font =:= color);
```

⟨*global* Efuns.displayname 31c⟩≡ (? 0—1)
```
  let displayname = ref ""
```

⟨*constant* Efuns.check 31d⟩≡ (? 0—1)
```
  let check = ref false
```

⟨main() *command line options* 31e⟩+≡ (29c) ◁31a 153a▷
```
  "-check", Arg.Set Globals.check, ": only for testing";
```

4.3 The default editor

The editor record is the global singleton that owns all shared state: open buffers, open files, the list of top windows, the global keymap, and color/font registries. Colors and fonts are stored in fixed-size arrays (256 entries) because make_attrX packs a color index and a font index into a single byte each. The edt_mutex protects this shared state from concurrent access by background threads (e.g., shell commands launched with M-!).

⟨main() *initial editor* 31f⟩≡ (29c)
```
  let editor = { Efuns.
      edt_buffers = Hashtbl.create 13;
      edt_files = Hashtbl.create 13;

      top_windows = [];

      edt_width = !!width;
      edt_height = !!height;
      edt_fg = !!foreground;
      edt_bg = !!background;
      edt_font = !!Globals.font;

      edt_map = Keymap.create ();
      edt_dirname = Sys.getcwd ();

      edt_vars = Store.new_store ();

      edt_colors_names = Array.make 256 "";
      edt_colors = Hashtbl.create 37;
      edt_colors_n = 0;

      edt_fonts_names = Array.make 256 "";
      edt_fonts = Hashtbl.create 37;
      edt_fonts_n = 0;

      edt_mutex = Mutex.create ()
  } in
```

4.4 Startup hooks

Rather than hardwiring initialization into `main`, efuns uses a start hooks pattern: each module (major modes, keybinding configurations, etc.) registers a closure via `add_start_hookX` at toplevel evaluation time. When `init_efunsX` runs, it drains the accumulated list and executes every hook. Crucially, it also clears `start_hooks` so hooks registered later (e.g., by code loaded from `.efunsrc`) can be collected and run in a second pass.

⟨`main()` *initialize the world* 32a⟩≡ (29c)
```
init_efuns editor (* launch first hooks *);
```

⟨*function* `Efuns.init` 32b⟩≡ (? 0—1)
```
let init_efuns editor =
  Globals.global_editor := Some editor;
  let hooks = List.rev !Hooks.start_hooks in
  Hooks.start_hooks := [];
  hooks |> List.iter (fun f -> f ())
```

⟨*constant* `Efuns.start_hooks` 32c⟩≡ (? 0—1)
```
(* Les hooks de lancement apres le chargement d'un module *)
let start_hooks = (ref []: (unit -> unit) list ref)
```

⟨*function* `Efuns.add_start_hook` 32d⟩≡ (? 0—1)
```
let add_start_hook hook =
  start_hooks := hook :: !start_hooks
```

⟨*toplevel* `Config` *starting hook* 32e⟩≡ (? 0—1)
```
let _ =
  Hooks.add_start_hook (fun () ->
    ⟨Config._5 start hooks options 162a⟩
    init_global_map ();
    Hooks.add_hook Top_window.handle_key_start_hook Misc_features.check_file;
  )
```

4.5 The graphical top window

The graphical layer is deliberately isolated behind a backend interface (`Xdraw`), so the core editor logic is portable across GTK, X11, or any future toolkit. `Graphics_efuns.initX` is the non-portable entry point: it opens the display, creates a `Top_window`, wires up the backend, performs an initial `update_display`, and enters the event loop.

⟨`main()` *run the UI* 32f⟩≡ (29c)
```
Graphics_efuns.init caps !initial_files;
```

⟨*function* `Efuns.backend` 32g⟩≡ (? 0—1)
```
let backend top_window =
  match top_window.graphics with
    None -> raise Not_found
  | Some x -> x
```

4.5.1 `Graphics.init()`

The actual `Graphics_efuns.init` function is backend-specific and therefore not shown here directly, but the pseudocode in the comments below outlines the universal pattern: open the graphics system, create a `Top_windowX`, attach a backend context via the `graphics` field, call `update_displayX` to paint the initial screen, and enter the event loop that dispatches to `Top_window.handlerX`.

4.5.2 `Top_window.create()`

Creating a top window involves a mutual-recursion puzzle: a `top_window` needs an active frame, but a frame's `win_up` field needs to point back to the `top_window`. The solution is `Window.create_at_topX`, which uses OCaml's `let rec` to build a window with a temporary self-referencing `win_up`, then `Top_window.create` patches it to `TopWindow top_window` after construction. Note the `-1` on height: one row is reserved for the minibuffer, which is not pre-created but allocated lazily on the first `M-x`.

⟨*function* `Top_window.create` 33a⟩≡ (? 0—1)

```
let create (caps : <frame_caps; ..>) =
  let edt = Globals.editor() in
  let buf = Ebuffer.default "*help*" in
  (* keep one line for the minibuffer, hence the -1 *)
  let window = Window.create_at_top  0 0 edt.edt_width (edt.edt_height - 1) in
  let frame = Frame.create_without_top caps window None buf in
  let top_window =
    {
      top_width = edt.edt_width;
      top_height = edt.edt_height;
      window = window;
      top_active_frame = frame;

      top_name = buf.buf_name;

      top_mini_buffers = [];
      top_second_cursor = None;

      graphics = None;
    }
  in

  (* adjust what Window.create_at_top could not do *)
  frame.frm_window.win_up <- TopWindow top_window;
  edt.top_windows <- top_window :: edt.top_windows;

  top_window
```

⟨*function* `Ebuffer.default` 33b⟩≡ (? 0—1)

```
let default name =
  try
    Hashtbl.find (Globals.editor()).edt_buffers name
  with Not_found ->
    let str =
      if name = "*help*"
      then help_buffer_content
      else ""
    in
    create name None (Text.create str) (Keymap.create ())
```

⟨*function* `Ebuffer.find_buffer_opt` 33c⟩≡ (? 0—1)

```
let find_buffer_opt name =
  try Some (Hashtbl.find (Globals.editor()).edt_buffers name)
  with Not_found -> None
```

⟨*constant* `Ebuffer.help_buffer_content` 33d⟩≡ (? 0—1)

```
let help_buffer_content =
"Welcome to Efuns, a small demo editor written in Ocaml.

Fabrice Le Fessant
PARA/SOR Project
```

```
  INRIA Rocquencourt
  "
```

⟨*function* `Window.create_at_top` 34a⟩≡ (? 0—1)

```
  let create_at_top xpos ypos width height =
    let rec window = {
      win_xpos = xpos;
      win_ypos = ypos;
      win_width = width;
      win_height = height;

      win_down = NoFrame;
      win_up = Window window; (* this will be adjusted in the caller *)

      win_mini = false;
    } in
    window
```

4.5.3 `Top_window.update_display()`

The redisplay engine is invoked after every keystroke and during initialization. It iterates over all top windows, walks the window tree with `Window.iterX` to repaint each frame via `Frame.displayX`, then separately repaints the active minibuffer (if any), restores the cursor, and finally tells the graphics backend to flush its updates to screen. The `\[@@profiling\]` annotation reflects that this is a hot path worth measuring.

⟨*function* `Top_window.update_display` 34b⟩≡ (? 0—1)

```
  let update_display () =
    (Globals.editor()).top_windows |> List.iter (fun top_window ->
       top_window.window |> Window.iter (fun frm ->
         Frame.display top_window frm
       );
       (match top_window.top_mini_buffers with
        | [] -> ()
        | frm :: _ -> Frame.display top_window frm
       );
       cursor_on top_window;
       let graphic = Efuns.backend top_window in
       graphic.Xdraw.update_display();
    )
  [@@profiling]
```

4.6 The event loop

The event loop is provided by the graphics backend (GTK or X11). On each event, `Top_window.handler` acquires the editor mutex (necessary because external processes like `M-!` run in separate threads) and dispatches based on event type. For key presses, `handle_key` is the critical path: it turns off the cursor, runs `handle_key_start_hook`, computes the modifier (Control, Meta, or both), looks up the binding via `Keymap.get_binding` (cascading through minor → major → buffer → global maps), executes the action, runs `handle_key_end_hook`, and finally calls `update_display` to repaint all affected frames.

4.6.1 `Event`

The event type (`Xtypes.event`) is defined in the graphics abstraction layer and is not shown here. The key variant is `XTKeyPress`, which carries modifier bits and a keysym integer.

4.6.2 Top_window.handler()

Every event is handled under the editor mutex via `with_lockX`, which uses `Fun.protect` to guarantee the lock is released even if an exception escapes. The handler simply pattern-matches on the event type and dispatches; currently only `XTKeyPress` is shown, but resize and mouse events follow the same structure.

⟨*function* Top_window.handler 35a⟩≡ (? 0—1)
```
let handler top_window event =
  Globals.with_lock (fun () ->
    match event with
    ⟨Top_window.handler() match event cases 35c⟩
  )
```

⟨*function* Efuns.with_lock 35b⟩≡ (? 0—1)
```
let with_lock f =
  let edt = editor () in
  Mutex.lock edt.edt_mutex;
  Fun.protect ~finally:(fun () -> Mutex.unlock edt.edt_mutex) f
```

4.6.3 Top_window.handle_key()

The `handle_key` function is the heart of the keystroke pipeline. It first saves the raw keysym in `keypressed` (used by `self_insert_commandX` to avoid defining a separate action for every printable character), then turns off the cursor and clears any lingering message from the minibuffer. The start hook runs per-buffer (via `Var.get_var`) so modes like eshell can intercept keystrokes, while the end hook is global. Between the hooks, `try_mapX` does the actual keymap lookup and action execution. Finally, `update_display` repaints everything—meaning every single keystroke triggers a full redisplay pass.

⟨Top_window.handler() *match event cases* 35c⟩≡ (35a) 85b▷
```
| Xtypes.XTKeyPress (modifiers, _s, keysym) ->
    handle_key_and_macro top_window modifiers keysym
```

⟨*function* Top_window.handle_key 35d⟩≡ (? 0—1)
```
let handle_key top_window modifiers keysym =
  keypressed := keysym;
  let frame = top_window.top_active_frame in
  let buf = frame.frm_buffer in

  clean_display (); (* set cursor off *)
  clear_message top_window;

  Hooks.exec_hooks(try Var.get_var buf handle_key_start_hook with _ ->[]) frame;

  let mod_ =
    ⟨Top_window.handle_key() compute mod 36e⟩
  in
  let key = (mod_, keysym) in
  begin
    try
      (* should lead to an action being triggered and modifying things! *)
      try_map frame key
    with
    ⟨Top_window.handle_key() handle exception of try_map 82b⟩
  end;

  Hooks.exec_hooks (try Var.get_global handle_key_end_hook with _ -> []) ();

  update_display () (* will set cursor back on and many other things *)
```

⟨*constant* `Top_window.keypressed` 36a⟩≡ (? 0—1)

```
let keypressed = ref 0
```

⟨*constant* `Top_window.handle_key_start_hook` 36b⟩≡ (? 0—1)

```
let handle_key_start_hook = Store.create_abstr "handle_key_start_hook"
```

⟨*constant* `Top_window.handle_key_end_hook` 36c⟩≡ (? 0—1)

```
let handle_key_end_hook = Store.create_abstr "handle_key_end_hook"
```

⟨*constant* `Top_window.meta` 36d⟩≡ (? 0—1)

```
let meta = ref Xtypes.mod1Mask
```

⟨`Top_window.handle_key()` *compute mod* 36e⟩≡ (35d)

```
let mask = Xtypes.controlMask lor !meta in
let diff = modifiers land mask in
match () with
| _ when diff =|= 0 -> NormalMap
| _ when diff =|= Xtypes.controlMask -> ControlMap
| _ when diff =|= !meta -> MetaMap
| _ -> ControlMetaMap
```

Chapter 5

Text Management

This chapter covers the internals of `Text.t`—the gap buffer implementation that is the foundation of all editing operations. The key insight is that a gap buffer makes sequential editing at the cursor position O(1), at the cost of O(n) for random-position edits (moving the gap). Since users typically type sequentially, this is an excellent trade-off. The complexity lies in keeping everything consistent: positions must be translated between "user" and "internal" coordinates, all registered points must be updated when the gap moves, and the `text_newlines` array must be maintained after every insertion or deletion.

5.1 The gap buffer

The gap buffer is a classic data structure for text editors, also used by GNU Emacs. The idea is to maintain a single contiguous byte array (`text_string`) with an empty "hole" (the gap) at the cursor position. Inserting a character at the cursor is O(1)—just write into the gap and shrink it. Inserting elsewhere requires first moving the gap to that position, which costs O(n) in the distance moved but is cheap for sequential edits. The key fields are `gpoint` (where the gap starts), `gsize` (how many unused bytes the gap contains), and `text_size` (the total allocation including the gap). The logical file size is `text_size - gsize`. Alternatives like piece tables or ropes exist, but gap buffers are simple and well-suited to the sequential nature of typing.

⟨`Text.t` *gap fields* 37a⟩≡ (13a)
```
(* g for gap *)
mutable gpoint : point;
mutable gsize : int;
```

⟨*function* `Text.size` 37b⟩≡ (? 0—1)
```
let size text =
  text.text_size - text.gsize
```

⟨*function* `Text.get_char` 37c⟩≡ (? 0—1)
```
let get_char text point =
  let pos = point.pos in
  let gpos = text.gpoint.pos in
  let pos =
    if pos = gpos
    then pos + text.gsize
    else pos
  in
  if pos < text.text_size
  then Bytes.get text.text_string pos
  else '\000'
```

5.2 Positions

The gap buffer creates a fundamental distinction between two position systems. User positions are contiguous character offsets as the user sees them (0 to file length). Internal positions (`position2`) are offsets into `text_string` that include the gap—so a position after the gap is shifted by `gsize`. `Text.get_position` converts internal → user (subtracting `gsize` if past the gap), and `Text.set_position` does the reverse.

5.2.1 User positions

A "user position" is what `C-x =` displays: a contiguous offset from 0 to the logical file length, as if the gap did not exist. `Text.get_position` converts from the internal `point.pos` to a user position by subtracting `gsize` when the point lies past the gap.

⟨*misc keys* 38a⟩+≡ (25) ◁27d 62a▷
```
[c_x; NormalMap, Char.code '='], Misc_features.cursor_position;
```
Uses `Efuns.mod_ident.ControlMap`.

⟨*function* `Complex.get_pos` 38b⟩≡ (? 0—1)
```
let describe_position frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Message.message frame
    (Printf.sprintf "Char position %d" (Text.get_position text point))
[@@interactive]
```

⟨*function* `Misc.cursor_position` 38c⟩≡ (? 0—1)
```
let cursor_position frm =
  let (_, text, point) = Frame.buf_text_point frm in
  let char = Text.get_char text point in
  let coord = Text.point_coord text point in
  Message.message frm
    (Printf.sprintf "Char: '%c' (%d, #o%o, #x%x) point=%d line=%d column=%d"
       char
       (Char.code char) (Char.code char) (Char.code char)
       (Text.get_position text point)
       (coord.Text.c_line + 1)
       coord.Text.c_col
    )
[@@interactive]
```

⟨*function* `Text.get_position` 38d⟩≡ (? 0—1)
```
let get_position text point =
  (* gap handling *)
  if point.pos > text.gpoint.pos
  then point.pos - text.gsize
  else point.pos
```

5.2.2 Internal positions

Internal positions (`position2`) are raw offsets into `text_string`, so they include the gap. `Text.set_position` converts a user position to an internal one by adding `gsize` when the user position is past `gpoint.pos`. Most code outside `Text` works with user positions; internal positions appear only in the gap management internals.

⟨*type* `Text.position2` 38e⟩≡ (? 0—1)
```
type position2 = int
```

⟨*function* `Text.set_position` 39a⟩≡ (? 0—1)
```
let set_position text point pos =
  move_point_to_pos text point
    (if pos > text.gpoint.pos
     then pos + text.gsize
     else pos
     )
```

5.3 Points

Points are mutable cursors registered with a `Text.t`. Whenever the gap moves or the buffer is modified, all registered points are adjusted to maintain their logical position in the text. This is why points must be created via `Text.new_point` (which adds them to `text_points`) and cleaned up via `Text.remove_point`. The `with_dup_point` helper ensures cleanup even on exceptions—a critical resource management pattern since leaked points would be updated forever, wasting work and potentially causing bugs.

⟨`Text.t` *other fields* 39b⟩≡ (13a) 110a▷
```
mutable text_points : point list;
```

⟨*function* `Text.new_point` 39c⟩≡ (? 0—1)
```
let new_point text =
  let p = { pos = 0; line = 0; } in
  text.text_points <- p :: text.text_points;
  p
```

⟨*function* `Text.dup_point` 39d⟩≡ (? 0—1)
```
let dup_point text point =
  let p = { pos = point.pos; line = point.line } in
  text.text_points <- p :: text.text_points;
  p
```

⟨*function* `Text.remove_point` 39e⟩≡ (? 0—1)
```
let remove_point text p =
  text.text_points <-
    text.text_points |> List.fold_left (fun points point ->
        if point == p
        then points
        else point :: points
    ) [];
  (* being defensive *)
  p.pos <- -1;
  p.line <- -1;
  ()
```

⟨*function* `Text.goto_point` 39f⟩≡ (? 0—1)
```
let goto_point _text p1 p2 =
  (* less: could assert they references the same text *)
  p1.pos <- p2.pos;
  p1.line <- p2.line
```

⟨*function* `Text.with_dup_point` 39g⟩≡ (? 0—1)
```
let with_dup_point text point f =
  let p = dup_point text point in
  Fun.protect ~finally:(fun () -> remove_point text p) (fun () -> f p)
```

⟨*function* `Text.with_new_point` 39h⟩≡ (? 0—1)
```
let with_new_point text f =
  let p = new_point text in
  Fun.protect ~finally:(fun () -> remove_point text p) (fun () -> f p)
```

5.4 Delta

A delta is a signed distance between two positions, in user coordinates. `Text.distance` (and the internal `low_distance`) must account for the gap: if the two positions straddle the gap, the raw difference `p2 - p1` would include the gap bytes, so `gsize` must be subtracted. The four cases in `low_distance` cover every combination of both positions being before, after, or straddling the gap.

⟨*type* `Text.delta` 40a⟩≡ (? 0—1)
```
type delta = int
```

⟨*function* `Text.distance` 40b⟩≡ (? 0—1)
```
let distance text p1 p2 =
  low_distance text p1.pos p2.pos
```

⟨*function* `Text.low_distance` 40c⟩≡ (? 0—1)
```
let low_distance text p1 p2 =
  if p1 >= p2
  then 0
  else
    if p1 <= text.gpoint.pos
    then
      if p2 <= text.gpoint.pos
      then p2 - p1
      else p2 - p1 - text.gsize
    else
      if p2 <= text.gpoint.pos
      then p2 - p1 + text.gsize
      else p2 - p1
```

5.5 Lines and columns

Beyond raw byte positions, the editor also needs line/column coordinates for the status line and for vertical movement commands. Each point carries a `line` field that is maintained alongside `pos` during every gap move and insertion. Computing the column requires gap awareness: if the gap sits between the beginning of the line and the point, `gsize` must be subtracted from the raw offset difference. Note that coordinates here are logical (text line and column); screen coordinates (pixel x/y) are a separate concern handled later by the line representation and frame layers.

⟨*type* `Text.coord` 40d⟩≡ (? 0—1)
```
type coord = {
  c_col: int;
  c_line: int;
}
```

⟨*function* `Text.point_line` 40e⟩≡ (? 0—1)
```
let point_line text point =
  (* defensive: *)
  if (point.line <> find_line_of_pos text point.pos)
  then Logs.err (fun m -> m "TODO: point.line = %d != find_line ... = %d"
                    point.line (find_line_of_pos text point.pos));
  point.line
```

⟨*function* Text.point_col 41a⟩≡ (? 0—1)

```
let point_col text point =
  let pos = point.pos in
  let gpos = text.gpoint.pos in
  let bol = text.text_newlines.(point.line).position in
  (* gap handling *)
  if bol <= gpos && gpos < pos
  then pos - bol - text.gsize
  else pos - bol
```

⟨*function* Text.point_coord 41b⟩≡ (? 0—1)

```
let point_coord text point =
  { c_col = point_col text point;
    c_line = point_line text point;
  }
```

⟨*function* Text.move_point_to 41c⟩≡ (? 0—1)

```
let move_point_to_pos text point pos =
  point.pos <- pos;
  point.line <- find_line_of_pos text pos
```

⟨*function* Text.find_line_of_pos 41d⟩≡ (? 0—1)

```
let find_line_of_pos text pos =
  let gpos = text.gpoint.pos in
  let gline = text.gpoint.line in
  let gap_end = gpos + text.gsize in

  if pos >= gap_end then
    (* go forward *)
    let rec iter line =
      if line >= text.text_nlines
      then text.text_nlines - 1
      else
        if text.text_newlines.(line).position > pos
        then line - 1
        else iter (line + 1)
    in
    iter (gline+1)
  else
     (* go backward *)
     let rec iter line =
       if line > 0
       then
         if text.text_newlines.(line).position > pos
         then iter (line - 1)
         else line
      else 0
     in
     iter gline
[@@profiling]
```

⟨*function* Text.fmove_res 41e⟩≡ (? 0—1)

```
let fmove_res text p delta =
  if delta = 0 then 0 else
  let gpos = text.gpoint.pos in
  let size = text.text_size in
  let gap_end = gpos + text.gsize in
  let gline = text.gpoint.line in
  let pos = p.pos in
  let old_pos = pos in
```

```
    let lines = text.text_newlines in
    let rec iter y point end_point =
      if end_point > point
      then
        let end_line = lines.(y+1).position in
        if end_point >= end_line
        then iter (y+1) end_line end_point
        else (y, end_point)
      else (y,point)
    in
    let (y,pos) =
      if pos + delta <= gpos
      then iter p.line pos (pos+delta)
      else
        if pos >= gap_end
        then
          let delta = min delta (size - pos) in
          iter p.line pos (pos + delta)
        else
          let delta = min (delta - (gpos - pos)) (size - gap_end) in
          iter gline gap_end (gap_end + delta)
    in
    p.pos <- pos;
    p.line <- y;
    low_distance text old_pos pos
```

⟨*function* Text.bmove_res 42⟩≡ (? 0—1)

```
  let bmove_res text p delta =
    if delta = 0 then 0 else
    let gpos = text.gpoint.pos in
    let gap_end = gpos + text.gsize in
    let gline = text.gpoint.line in
    let pos = p.pos in
    let old_pos = pos in
    let lines = text.text_newlines in
    let rec iter y end_point =
      let start_line = lines.(y).position in
      if end_point >= start_line then
        (y, end_point)
      else
        iter (y-1) end_point
    in
    let (y,pos) =
      if pos - delta >= gap_end then
        if pos - delta < lines.(gline+1).position then
          (
            gline, pos - delta)
        else
          iter p.line (pos-delta)
      else
      if pos <= gpos then
        let delta = min delta pos in
        iter p.line (pos - delta)
      else
      let delta = min (delta - (pos - gap_end)) gpos in
      iter gline (gpos - delta)
    in
    p.pos <- pos;
    p.line <- y;
    low_distance text pos old_pos
```

⟨*function* `Text.bmove` 43a⟩≡ (? 0—1)
```
let bmove text p delta =
  bmove_res text p delta |> ignore
```

⟨*function* `Text.fmove` 43b⟩≡ (? 0—1)
```
let fmove text p delta =
  fmove_res text p delta |> ignore
```

⟨*function* `Text.move_res` 43c⟩≡ (? 0—1)
```
let move_res text point n =
  if n > 0
  then fmove_res text point n
  else bmove_res text point (-n)
```

⟨*function* `Text.move` 43d⟩≡ (? 0—1)
```
let move text point n =
  move_res text point n |> ignore
```

5.6 Inserting characters

Inserting text into the gap buffer is a multi-step process: (1) move the gap to the insertion point (which requires blitting bytes and updating all points and line records), (2) extend the gap if the string to insert is larger than the remaining gap space (doubling the buffer), (3) copy the string into the gap, (4) update the `text_newlines` array if the string contains newlines, and (5) advance the gap position past the inserted text. Each step must carefully maintain consistency of all registered points.

⟨*function* `Text.low_insert` 43e⟩≡ (? 0—1)
```
let low_insert text pos str =
  let strlen = String.length str in
  ⟨Text.low_insert() fail if readonly buffer 150f⟩

  ⟨Text.low_insert() move gap to point 44a⟩
  (* subtle: don't move those 'let' earlier, because moving the gap do side
   * effects on the position and line of the gpoint.
   *)
  let gpos = text.gpoint.pos in
  let gline = text.gpoint.line in

  cancel_repr text gpos gline;
  ⟨Text.low_insert() extend gap if not enough space 45a⟩
  (* all points should have their point.pos correctly adjusted *)

  String.blit  str 0   text.text_string gpos    strlen;
  (* todo: should refontify! *)
  Array.fill text.text_attrs gpos strlen direct_attr;

  let (nbr_newlines, _nbr_chars) = Utils.count_char str '\n' in
  if nbr_newlines > 0 then
  begin
    ⟨Text.low_insert() adjust newlines when str contains newlines 45d⟩
  end;
  let gline = text.gpoint.line in
  text.text_points |> List.iter (fun p ->
    if p.pos > gpos
      (* bugfix: there was extra condition line = gline. But bug! unit test *)
    then p.line <- p.line + nbr_newlines;

  );
```

```
  text.gpoint <- { pos = gpos + strlen; line = gline + nbr_newlines };
  text.gsize <- text.gsize - strlen;
  (gpos, strlen, text.text_modified)
```

5.6.1 Moving the gap

Moving the gap is the most expensive part of an insertion. `move_gpoint_to` blits the bytes between the old gap position and the target position, copies the corresponding attributes, adjusts `text_newlines` entries for any lines that crossed the gap boundary, and shifts every registered point whose internal position falls in the moved region. The two cases (moving the gap forward vs. backward) are symmetric but use opposite blit directions.

⟨`Text.low_insert()` *move gap to point* 44a⟩≡ (46b 43e)
```
move_gpoint_to text pos;
```

⟨*function* `Text.move_gpoint_to` 44b⟩≡ (? 0—1)
```
let move_gpoint_to text pos =
  let gpos = text.gpoint.pos in
  let gsize = text.gsize in
  let gline = text.gpoint.line in
  let gap_end = gpos + gsize in

  ⟨Text.move_gpoint_to() reset text_clean 110b⟩
  if pos <> gpos then
    if pos < gpos then begin
      ⟨Text.move_gpoint_to() when pos is before gpos 44c⟩
    end else begin
      ⟨Text.move_gpoint_to() when pos is after gpos 44d⟩
      end
[@@profiling]
```

⟨`Text.move_gpoint_to()` *when pos is before gpos* 44c⟩≡ (44b)
```
let delta = gpos - pos in
let (delta_line,_) =
  Utils.count_char_sub (Bytes.to_string text.text_string) pos delta '\n'
in
Bytes.blit text.text_string pos   text.text_string (pos + gsize)   delta;
Array.blit text.text_attrs  pos   text.text_attrs  (pos + gsize)   delta;
for i = gline - delta_line + 1 to gline do
  text.text_newlines.(i).position <- text.text_newlines.(i).position + gsize
done;
text.text_points |> List.iter (fun p ->
  if p.pos > pos && p.pos <= gpos
  then p.pos <- p.pos + gsize
);
text.gpoint <- { pos = pos; line = gline - delta_line };
```

⟨`Text.move_gpoint_to()` *when pos is after gpos* 44d⟩≡ (44b)
```
let delta = pos - gap_end in
let (delta_line,_) =
  Utils.count_char_sub (Bytes.to_string text.text_string) gap_end delta '\n' in
Bytes.blit text.text_string gap_end   text.text_string gpos   delta;
Array.blit text.text_attrs  gap_end   text.text_attrs  gpos   delta;
for i = gline + 1 to gline + delta_line do
  text.text_newlines.(i).position <- text.text_newlines.(i).position - gsize
done;
text.text_points |> List.iter (fun p ->
    if p.pos >= gap_end && p.pos <= pos
    then p.pos <- p.pos - gsize
);
text.gpoint <- { pos = pos - gsize; line = gline + delta_line }
```

5.6.2 extending the gap

When the gap is too small for the string being inserted, `extend_gap` allocates a new, larger `text_string` (and `text_attrs`) and copies the content around the gap into the new buffer, widening the gap by at least `add_amount` bytes (default 200). All `text_newlines` positions past the gap and all registered point positions must be shifted by the added size.

⟨`Text.low_insert()` *extend gap if not enough space* 45a⟩≡ (43e)
```
if strlen > text.gsize
then extend_gap text strlen;
```

⟨*constant* `Text.add_amount` 45b⟩≡ (? 0—1)
```
let add_amount = define_option ["add_amount"] "Size of the gap in the buffer"
  int_option 200
```

⟨*function* `Text.extend_gap` 45c⟩≡ (? 0—1)
```
let extend_gap text amount =
  let add_size = max !!add_amount
      ((amount / !!add_amount) * !!add_amount + !!add_amount) in
  let old_size = text.text_size in
  let gsize = text.gsize in

  let gpos = text.gpoint.pos in
  let gap_end = gpos + gsize in

  let new_text = Bytes.create (old_size + add_size) in
  Bytes.blit  text.text_string 0   new_text 0    gpos;
  Bytes.blit  text.text_string gap_end   new_text (gap_end + add_size)
    (old_size - gap_end);
  text.text_string <- new_text;

  let new_attrs = Array.make (old_size + add_size) direct_attr in
  Array.blit   text.text_attrs 0   new_attrs 0   gpos;
  Array.blit   text.text_attrs gap_end    new_attrs (gap_end + add_size)
    (old_size - gap_end);
  text.text_attrs <- new_attrs;

  for i = text.gpoint.line + 1 to text.text_nlines - 1 do
    text.text_newlines.(i).position <-
      text.text_newlines.(i).position + add_size
  done;
  text.text_points |> List.iter (fun p ->
      if p.pos > gpos
      then p.pos <- p.pos + add_size
  );
  text.gsize <- gsize + add_size;
  text.text_size <- old_size + add_size;
  ()
[@@profiling]
```

5.6.3 Adjusting newlines information

When the inserted string contains newlines, the `text_newlines` array must be updated: existing entries after the insertion point are shifted right by `nbr_newlines` slots, and the new line positions are filled in by scanning the inserted text for `'\\n'` characters. If the array is too small, it is grown first. This is the bookkeeping that makes `point_line` and column calculations correct after multi-line insertions.

⟨`Text.low_insert()` *adjust newlines when str contains newlines* 45d⟩≡ (43e)
```
if (Array.length text.text_newlines - text.text_nlines) < nbr_newlines then
```

```
    begin
      ⟨Text.low_insert() grow newlines 46a⟩
    end;

  Array.blit
    text.text_newlines (gline+1)
    text.text_newlines (gline+1+ nbr_newlines)
   (text.text_nlines - gline -1);
  text.text_nlines <- text.text_nlines + nbr_newlines;

  (* similar to compute_newlines() but just for lines after gpos *)
  let rec iter n pos =
    let new_pos =
     Bytes.index_from text.text_string pos '\n' in
    text.text_newlines.(gline+n) <- mk_line_with_pos (new_pos + 1);
    if n < nbr_newlines
    then iter (n+1) (new_pos + 1)
  in
  iter 1 gpos;
```

⟨Text.low_insert() *grow newlines* 46a⟩≡ (45d)

```
  let old_size = text.text_nlines in
  let new_cache = Array.make (old_size + (max 20 nbr_newlines))
    { (mk_line_with_pos (-1)) with line_modified = false }
  in
  Array.blit
    text.text_newlines 0
    new_cache 0
    old_size;
  text.text_newlines <- new_cache;
```

5.7 Deleting characters

Deletion is the mirror of insertion: after moving the gap to the deletion point, we simply expand `gsize` by the number of bytes to delete—the "deleted" bytes are swallowed into the gap. Any points that were inside the deleted region are snapped back to `gpos`, and all points after the deletion have their `line` decremented by the number of newlines that were removed. The actual bytes are not zeroed; they become part of the gap and will be overwritten by future insertions or cleaned up lazily by `clean_text`.

⟨*function* Text.low_delete 46b⟩≡ (? 0—1)

```
  let low_delete text pos len =
    ⟨Text.low_insert() fail if readonly buffer 150f⟩

    ⟨Text.low_insert() move gap to point 44a⟩
    (* subtle: don't move those 'let' earlier, because moving the gap do side
     * effects on the position and line of the gpoint.
     *)
    let gpos = text.gpoint.pos in
    let gline = text.gpoint.line in

    cancel_repr text gpos gline;

    let gap_end = gpos + text.gsize in
    let len = min (text.text_size - gap_end) len in

    let str = Bytes.sub_string text.text_string gap_end len in

    let (nbr_newlines, _nbr_chars) = Utils.count_char str '\n' in
```

```
    if nbr_newlines > 0 then begin
      ⟨Text.low_delete() adjust newlines when str contained newlines 47a⟩
    end;
    text.text_points |> List.iter (fun p ->
        if p.pos > gap_end + len
        then p.line <- p.line - nbr_newlines
        else
          (* points that were in deleted region *)
          if p.pos > gpos then begin
             p.pos <- gpos;
             p.line <- gline
           end
    );
    text.gsize <- text.gsize + len;
    (gpos, str, text.text_modified)
```

⟨Text.low_delete() *adjust newlines when str contained newlines* 47a⟩≡ (46b)

```
  Array.blit
    text.text_newlines (gline + nbr_newlines + 1)
    text.text_newlines (gline + 1)
   (text.text_nlines - gline - nbr_newlines - 1);
  text.text_nlines <- text.text_nlines - nbr_newlines;
```

5.8 Attributes

Each character in the buffer carries an attribute that encodes its foreground color, background color, font, and highlight status into a single `int`. This packed representation—8 bits for foreground, 8 bits for background, 8 bits for font, and 1 bit for highlighting—is critical for efficiency because the `text_attrs` array has one entry per character. Colors and fonts are stored as indices into global maps (`edt_colors`, `edt_fonts`) rather than as strings, which keeps attribute comparison and storage cheap. The first two color indices (0 and 1) are conventionally reserved for foreground and background, echoing the stdin/stdout/stderr numbering convention.

⟨*type* Text.attribute 47b⟩≡ (? 0—1)

```
  type attribute = int
```

⟨Text.t *attribute fields* 47c⟩≡ (13a)

```
  mutable text_attrs : attribute array;
```

⟨*function* Text.make_attr 47d⟩≡ (? 0—1)

```
  let make_attr fg bg font highlighted =
    let attr =
       fg +
       (bg lsl 8) +
       (font lsl 16) +
       (if highlighted then highlight_bit else 0)
    in
    attr
```

⟨*constant* Text.highlight_bit 47e⟩≡ (? 0—1)

```
  let highlight_bit = 1 lsl 24
```

⟨*constant* Text.direct_attr 47f⟩≡ (? 0—1)

```
  let direct_attr =  make_attr 0 1 0 false
```

⟨*constant* Text.inverse_attr 47g⟩≡ (? 0—1)

```
  let inverse_attr =  make_attr 1 0 0 false
```

⟨*function* Text.get_attr 48a⟩≡ (? 0—1)
```
let get_attr text point =
  let pos =
    (* gap handling *)
    if point.pos = text.gpoint.pos
    then point.pos + text.gsize
    else point.pos
  in
  if pos < text.text_size
  then text.text_attrs.(pos)
  else direct_attr
```

⟨*function* Text.set_attr 48b⟩≡ (? 0—1)
```
let set_attr text point attr =
  let y = point.line in
  let pos =
    (* gap handling *)
    if point.pos = text.gpoint.pos
    then point.pos + text.gsize
    else point.pos
  in
  if pos < text.text_size then begin
    cancel_repr text pos y;
    text.text_attrs.(pos) <- attr
  end
```

⟨*function* Text.set_attrs 48c⟩≡ (? 0—1)
```
let set_attrs text point len attr = (* should not exceed one line *)
  if len > 0 then
    let gap_end = text.gpoint.pos + text.gsize in

    cancel_repr text point.pos point.line;

    let pos = point.pos in
    let gpos = text.gpoint.pos in
    let before, after, after_pos =
      if pos > gap_end then
        0, (min (text.text_size - pos) len), pos
      else
      if pos + len <= gpos then
        0, len, pos
      else
      let before = gpos - pos in
      let after = min (len - before) (text.text_size - gap_end) in
      before, after, gap_end
    in
    if before > 0
    then Array.fill text.text_attrs pos before attr;
    Array.fill text.text_attrs after_pos after attr
```

⟨*function* Text.unset_attrs 48d⟩≡ (? 0—1)
```
let unset_attrs text =
  Array.fill text.text_attrs 0 (Array.length text.text_attrs) direct_attr;
  text.text_newlines |> Array.iter (fun line -> line.line_modified <- true)
```

5.8.1 Colors

Colors are stored as indices into a global palette (`edt_colors`) rather than as color name strings. `Window.get_color` lazily allocates an index the first time a color name is seen, up to a maximum of 256. By convention, index 0 is

the foreground color and index 1 is the background, established during initialization—a numbering convention reminiscent of stdin/stdout/stderr.

⟨Efuns.editor *other fields* 49a⟩+≡ (20a) ◁26c 52c▷
```
edt_colors_names : string array;
edt_colors : (string,int) Hashtbl.t;
mutable edt_colors_n : int;
```

⟨*function* Window.get_color 49b⟩≡ (? 0—1)
```
let get_color color_name =
  let edt = Globals.editor() in
  try
    Hashtbl.find edt.edt_colors color_name
  with Not_found ->
    if edt.edt_colors_n = 256
    then raise Not_found
    else begin
      let n = edt.edt_colors_n in
      edt.edt_colors_n <- n + 1;
      edt.edt_colors_names.(n) <- color_name;
      Hashtbl.add edt.edt_colors color_name n;
      n
    end
```

⟨main() *misc initializations* 49c⟩≡ (29c) 52e▷
```
(* color 0 is foreground *)
Attr.get_color !!foreground |> ignore;
(* color 1 is background *)
Attr.get_color !!background |> ignore;
(* color 2 is highlight *)
Attr.get_color !!highlight_color |> ignore;
```

The `Simple.colorX` function implements batch syntax coloring: it iterates through the buffer searching for a regexp, and for each match sets the attribute on the matched region. The `strict` flag implements implicit word boundaries by checking the characters before and after the match against the buffer's `buf_syntax_table`. This is not incremental like Emacs's `font-lock-mode`—it recolorizes the entire buffer each time—but it is simple and sufficient for many uses.

⟨*function* Simple.color 49d⟩≡ (? 0—1)
```
let color buf regexp strict attr =
  let text = buf.buf_text in
  Text.with_new_point text (fun point ->
  try
    while true do
      let len = Text.search_forward text regexp point in
      let before =
        if Text.bmove_res text point 1 = 1 then begin
          let c = Text.get_char text point in
          Text.fmove text point (len+1);
          c
        end else begin
          let c = Text.get_char text point in
          Text.fmove text point (len+1);
          c
        end
      in
      let after = Text.get_char text point in
      if not (strict && (buf.buf_syntax_table.(Char.code before) ||
                         buf.buf_syntax_table.(Char.code after))) then
        begin
```

```
          Text.bmove text point len;
          Text.set_attrs text point len attr;
          Text.fmove text point len;
          ()
        end
    done
  (* at some point Text.search_forward will return Not_found *)
  with Not_found ->
    buf.buf_modified <- buf.buf_modified + 1
  )
[@@profiling]
```

5.8.2 Highlights

Highlighting uses the spare bit 24 in the attribute integer (`highlight_bit`). `Simple.highlight_regionX` ORs this bit onto every character in a range; `Simple.unhighlight_regionX` ANDs it off. The highlighting is deliberately transient: `handle_key_start_hook` clears it on every keystroke, so the selection vanishes the moment the user types—matching Emacs's "transient mark mode" behavior. A separate `highlighted_chars` list tracks per-character highlights (used by `paren_mode` to flash the matching parenthesis), which are also cleaned up on the next key event.

⟨*constant* `Main.highlight_color` 50a⟩≡ (? 0—1)

```
let highlight_color = define_option ["highlight_color"] "" color_option "cyan"
```

⟨*function* `Simple.highlight_region` 50b⟩≡ (? 0—1)

```
let highlight_region buf debut fin =
  let text = buf.buf_text in
  Text.with_new_point text (fun curseur ->
  Text.with_new_point text (fun final ->
    Text.set_position text curseur debut;
    Text.set_position text final fin;
    while curseur < final do
      let attr = Text.get_attr text curseur in
      Text.set_attr text curseur (attr lor Text.highlight_bit);
      Text.fmove text curseur 1
    done;
    buf.buf_modified <- buf.buf_modified + 1
  ))
```

⟨*function* `Simple.unhighlight_region` 50c⟩≡ (? 0—1)

```
let unhighlight_region buf debut fin =
  let text = buf.buf_text in
  Text.with_new_point text (fun curseur ->
  Text.with_new_point text (fun final ->
    Text.set_position text curseur debut;
    Text.set_position text final fin;
    while curseur < final do
      let attr = Text.get_attr text curseur in
      Text.set_attr text curseur (attr land (lnot Text.highlight_bit));
      Text.fmove text curseur 1;
    done;
    buf.buf_modified <- buf.buf_modified + 1
  ))
```

⟨*constant* `Simple.highlighted` 50d⟩≡ (? 0—1)

```
(* hightlighting of a region *)
let highlighted = ref None
```

⟨*function* `Simple.highlight` 51a⟩≡ (? 0—1)

```
let highlight frame =
  let frame =
    match !H.highlighted with
    | None -> frame
    | Some (frame, _d, _f) -> frame
  in
  let (buf, text, point) = Frame.buf_text_point frame in
  let mark = Ebuffer.get_mark buf point in
  let pt_debut, pt_fin =
    if point < mark
    then point, mark
    else mark, point
  in
  let pos1 = Text.get_position text pt_debut in
  let pos2 = Text.get_position text pt_fin in
  let pos_debut_to_hl, pos_fin_to_hl =
    match !H.highlighted with
    | None -> pos1, pos2
    | Some (_frame, d, f) ->
        if pos1 > d
        then H.unhighlight_region buf d pos1;
        if pos2 < f
        then H.unhighlight_region buf pos2 f;
        if pos1 < d
        then pos1, d
        else
          if pos2 > f
          then f, pos2
          else pos1, pos1
  in
  H.highlighted := Some (frame, pos1, pos2);
  H.highlight_region buf pos_debut_to_hl pos_fin_to_hl
```

⟨*function* `Frame.buf_text_point` 51b⟩≡ (? 0—1)

```
let buf_text_point frame =
  let buf = frame.frm_buffer in
  let text = buf.buf_text in
  let point = frame.frm_point in
  (buf, text, point)
```

⟨*function* `Simple.unhighlight` 51c⟩≡ (? 0—1)

```
let unhighlight _frame =
  ⟨Simple.unhighlight() handle highlighted chars 52b⟩
  match !highlighted with
  | None -> ()
  | Some (frame,debut,fin) ->
     (*   if !keypressed <> XK.xk_Pointer_Drag1 then *)
     highlighted := None;
     let (buf, text, _) = Frame.buf_text_point frame in
     Hooks.exec_hooks (Var.get_global unhighlight_hook) (text, debut, fin);
     unhighlight_region buf debut fin
```

⟨*toplevel* `Highlight` *starting hook* 51d⟩≡ (? 0—1)

```
let _ =
  Hooks.add_start_hook (fun () ->
    Hooks.add_hook Top_window.handle_key_start_hook unhighlight;
  )
```

⟨`Simple.highlight_paren()` *remember highlighted chars* 51e⟩≡ (165g)

```
H.highlighted_chars := (buf,cursor,attr) :: !H.highlighted_chars;
```

⟨*constant* `Simple.highlighted_chars` 52a⟩≡ (? 0—1)
```
let highlighted_chars = ref []
```

⟨`Simple.unhighlight()` *handle highlighted chars* 52b⟩≡ (51c)
```
!highlighted_chars |> List.iter (fun (buf,curseur,old_attr) ->
  let text = buf.buf_text in
  (* less: could reset highlight bit instead of having to store old_attr *)
  Text.set_attr text curseur old_attr;
  buf.buf_modified <- buf.buf_modified + 1;
  Text.remove_point text curseur
);
highlighted_chars := [];
```

5.8.3 Fonts

⟨`Efuns.editor` *other fields* 52c⟩+≡ (20a) ◁49a 136c▷
```
edt_fonts : (string,int) Hashtbl.t;
edt_fonts_names : string array;
mutable edt_fonts_n : int;
```

⟨*function* `Window.get_font` 52d⟩≡ (? 0—1)
```
let get_font font_name =
  let edt = Globals.editor() in
  try
    Hashtbl.find edt.edt_fonts font_name
  with Not_found ->
    if edt.edt_fonts_n = 256
    then raise Not_found
    else begin
      let n = edt.edt_fonts_n in
      edt.edt_fonts_n <- n + 1;
      edt.edt_fonts_names.(n) <- font_name;
      Hashtbl.add edt.edt_fonts font_name n;
      n
    end
```

⟨`main()` *misc initializations* 52e⟩+≡ (29c) ◁49c
```
(* font 0 is initial font *)
Attr.get_font !!Globals.font |> ignore;
```

5.9 Line representation

Between the raw text bytes and the pixels on screen lies the line representation layer. A single buffer character may occupy multiple screen columns (tabs expand to variable-width spaces, control characters display as `^M`), so we need a mapping from text positions to display positions. This is done via `Text.compute_representationX`, which walks a line's characters, expands special characters using `buf_charreprs`, and groups consecutive characters that share the same attribute and character width into boxes. The result is a `repr_string` (the expanded display bytes) and a list of `box` records that describe contiguous runs. This two-level representation lets the rendering code issue one `draw_string` call per box instead of per character.

5.9.1 Line boxes

⟨`Text.line` *representation fields* 52f⟩≡ (13c) 53g▷
```
mutable boxes : box list; (* sorted in reverse; head is last box on the line *)
```

⟨*type* `Text.box` 53a⟩≡ (? 0—1)
```
and box =
  {
    box_col : int;        (* col of box *)
    box_len : int;        (* len of box in Text.t string *)
    mutable box_attr : int;    (* common attribute *)

    ⟨Text.box other fields 53h⟩
  }
```

5.9.2 Characters representation, handling `^M`

⟨`Efuns.buffer` *other fields* 53b⟩+≡ (15a) ◁26b 95b▷
```
mutable buf_charreprs : Text.charreprs; (* 256 array *)
```

⟨*type* `Text.charreprs` 53c⟩≡ (? 0—1)
```
type charreprs = string array
```

⟨`Ebuffer.create()` *buffer other fields setup* 53d⟩≡ (15f)
```
buf_charreprs = Array.init 256 (fun i -> String.make 1 (Char.chr i));
```

⟨`Ebuffer.create()` *adjust charreprs* 53e⟩≡ (15f) 151a▷
```
for i=0 to 25 do
  let s = Bytes.make 2 '^' in
  Bytes.set s 1 (Char.chr (97+i));
  buf.buf_charreprs.(i) <- (Bytes.to_string s);
done;
```

⟨*function* `Ebuffer.compute_representation` 53f⟩≡ (? 0—1)
```
let compute_representation buf n =
  Text.compute_representation buf.buf_text buf.buf_charreprs n
```

5.9.3 Line string representation

⟨`Text.line` *representation fields* 53g⟩+≡ (13c) ◁52f
```
mutable repr_string : bytes;
mutable repr_len : int;
```

⟨`Text.box` *other fields* 53h⟩≡ (53a) 53i▷
```
box_charsize : int; (* common size *)
box_size : int;
```

⟨`Text.box` *other fields* 53i⟩+≡ (53a) ◁53h
```
box_pos_repr : int;  (* pos of box in representation string *)
```

5.9.4 `Text.compute_representation()`

This is the core rendering function: it walks a line's characters, expands each one through `charreprs` (handling tabs and control characters), groups consecutive characters with the same attribute and character size into boxes, and stores the result in the line's `repr_string` and box list. Two cursors run in parallel: `text_cursor` over `text_string` (skipping the gap) and `repr_cursor` over `repr_string` (the expanded display string). The inner loop accumulates characters into the current box as long as the attribute and character size match; when

either changes, the box is closed and a new one started. The \[@@profiling\] annotation reflects that this function is on the critical path of every redisplay.

⟨*function* Text.compute_representation 54a⟩≡ (? 0—1)
```
(* On devrait reprendre la representation la ou elle est ... *)
let compute_representation text charreprs n =
  ⟨Text.compute_representation if n is too big, return dummy line 56c⟩
  else
    let line = text.text_newlines.(n) in
    if line.line_modified then begin

      repr_string := line.repr_string;
      repr_len    := Bytes.length line.repr_string;
      let boxes = ref [] in

      let text_cursor = ref line.position in
      let repr_cursor = ref 0 in

      let end_pos = text.text_newlines.(n+1).position - 1 in

      ⟨Text.compute_representation() locals 55a⟩

      ⟨Text.compute_representation() adjust text_cursor if in gap 55g⟩
      while !text_cursor < end_pos do
        ⟨Text.compute_representation() loop text_cursor to end_pos 54c⟩
      done;
      ⟨Text.compute_representation() adjust line fields after loop 55e⟩
    end;
    line
[@@profiling]
```

⟨Text.line other fields 54b⟩≡ (13c)
```
mutable line_modified : bool;
```

⟨Text.compute_representation() loop text_cursor to end_pos 54c⟩≡ (54a)
```
let charcode = Char.code (Bytes.get text.text_string !text_cursor) in
let charattr = text.text_attrs.(!text_cursor) in
let charrepr =
  ⟨Text.compute_representation() compute charrepr, special char 151c⟩
  else charreprs.(charcode)
in
let charsize = String.length charrepr in

let box_len = ref 0 in

while !text_cursor < end_pos &&
  begin
    let char_code = Char.code (Bytes.get text.text_string !text_cursor)
    in
    let char_attr = text.text_attrs.(!text_cursor) in
    char_repr :=
       ⟨Text.compute_representation() compute char_repr, special char 151d⟩
       else charreprs.(char_code);
    char_size := String.length !char_repr;

    !char_size = charsize && char_attr = charattr
  end
do
  ⟨Text.compute_representation() grow repr_string if needed 55d⟩
  String.blit !char_repr 0 !repr_string !repr_cursor charsize;
```

```
    repr_cursor := !repr_cursor + charsize;
    text_cursor := !text_cursor + 1;
    box_len := !box_len +1;
    ⟨Text.compute_representation() adjust text_cursor if reach gap 56a⟩
  done;

  let box = {
    box_col = !box_col_start;
    box_len = !box_len;
    box_attr = charattr;

    box_charsize = charsize;
    box_size = !box_len * charsize;
    box_pos_repr = !repr_start;
  } in
  repr_start := !repr_cursor;
  box_col_start := !box_col_start + !box_len;
  boxes := box :: !boxes;
```

⟨Text.compute_representation() *locals* 55a⟩≡ (54a) 55f▷

```
  let box_col_start = ref 0 in
  let repr_start = ref 0 in

  let char_repr = ref "" in
  let char_size = ref 0 in
```

⟨*constant* Text.repr_string 55b⟩≡ (? 0—1)

```
  let repr_string = ref (Bytes.of_string "")
```

⟨*constant* Text.repr_size 55c⟩≡ (? 0—1)

```
  let repr_len = ref 0
```

⟨Text.compute_representation() *grow* repr_string *if needed* 55d⟩≡ (54c)

```
  if !repr_cursor + charsize >= !repr_len then begin
      (* find a better heuristic to realloc the line string *)
      let new_len = !repr_len +
          (low_distance text end_pos !text_cursor) + charsize * 2
      in
      let new_repr = Bytes.create new_len in
      Bytes.blit !repr_string 0 new_repr 0 !repr_cursor;
      repr_string := new_repr;
      repr_len := new_len;
  end;
```

⟨Text.compute_representation() *adjust line fields after loop* 55e⟩≡ (54a)

```
  line.boxes <- !boxes;
  line.repr_string <- !repr_string;
  line.repr_len <- !repr_cursor;
  line.line_modified <- false;
```

Gap management

⟨Text.compute_representation() *locals* 55f⟩+≡ (54a) ◁55a

```
  let gpos = text.gpoint.pos in
  let gsize = text.gsize in
```

⟨Text.compute_representation() *adjust* text_cursor *if in gap* 55g⟩≡ (54a)

```
  if !text_cursor >= gpos && !text_cursor < gpos + gsize
  then text_cursor := !text_cursor + gsize;
```

⟨Text.compute_representation() *adjust* text_cursor *if reach gap* 56a⟩≡ (54c)
```
if !text_cursor = gpos
then text_cursor := gpos + gsize;
```

Dummy lines

⟨*constant* Text.dummy_line 56b⟩≡ (? 0—1)
```
let (dummy_line : line) =
  {
    position = max_int;
    boxes = [];
    repr_len = 0;
    repr_string = Bytes.of_string "";
    line_modified = true;
  }
```

⟨Text.compute_representation *if n is too big, return dummy line* 56c⟩≡ (54a)
```
if n >= text.text_nlines - 1 then begin
    dummy_line.position <- text.text_size;
    dummy_line
end
```

5.9.5 Text.cancel_repr()

⟨*function* Text.cancel_repr 56d⟩≡ (? 0—1)
```
let cancel_repr text _point n =
  let line = text.text_newlines.(n) in
  line.line_modified <- true
```

Chapter 6

Main Top Window

This chapter presents the code by following the visual structure of the application window from top to bottom: menus, then the editing area (sub-windows and frames), then the status line, and finally the minibuffer at the very bottom. This is also the order in which a user discovers the interface, making the code easier to follow than a bottom-up data-structure-first approach.

6.1 Menus

Menus are defined as configurable option lists of (label, action-name) pairs. The actual menu creation code (commented out below) used to call `add_menu` on the old X11 widget; now the backend handles rendering. What matters here is that every menu entry resolves to a named action, so the entire menu bar is user-customisable via `.efunsrc`.

⟨*toplevel* `Config` *menu settings* 57a⟩≡ (? 0—1)

```
let _ =
  ⟨Config file menu setup 57c⟩
  ⟨Config edit menu setup 62d⟩
  ⟨Config help menu setup 63a⟩
  ⟨Config buffers menu setup 118c⟩
```

6.1.1 File menu

The file menu covers the core file lifecycle: open, save, save-as, and exit. The interesting design point is that `load_buffer` and `write_buffer` both delegate to `Select.select_file_from_pwdX`, which opens a minibuffer with filename completion—so these menu entries and `C-x C-f` share the exact same code path.

⟨*constant* `Top_window.file_menu` 57b⟩≡ (? 0—1)

```
let file_menu = define_option ["file_menu"] "" (list_option string2_option) []
```

⟨`Config` *file menu setup* 57c⟩≡ (57a)

```
if !!Top_window.file_menu =*= [] then begin
  Top_window.file_menu =:= [
    "Open File", "load_buffer";
    "Save Buffer", "save_buffer";
    ⟨file menu entries 119d⟩
    "", "";
    "Quit", "exit";
  ]
  end;
```

Opening a file

⟨*loading keys* 58a⟩≡ (25) 58f▷
```
[c_x; ControlMap, Char.code 'f'], Multi_buffers.load_buffer;
```
Uses Efuns.mod_ident.NormalMap.

⟨*function* Complex.load_buffer 58b⟩≡ (? 0—1)
```
let load_buffer (frame : Frame.t) : unit =
  set_previous_frame frame;
  Select.select_file_from_pwd frame "Find file: " (fun str ->
    Frame.load_file frame.caps frame.frm_window str |> ignore
  )
[@@interactive]
```

⟨*function* Frame.load_file 58c⟩≡ (? 0—1)
```
let load_file (caps : < frame_caps; ..>) (window : Window.t) (filename : string) : t =
  let buf = Ebuffer.read filename (Keymap.create ()) in
  let frame = create caps window None buf in
  Hooks.exec_named_hooks !!change_buffer_hooks frame;
  status_name frame buf.buf_name;
  frame
```

⟨*function* Ebuffer.read 58d⟩≡ (? 0—1)
```
let read filename local_map =
  let edt = Globals.editor() in
  let filename = Utils.normal_name edt.edt_dirname filename in
  try
    Hashtbl.find edt.edt_files filename
  with Not_found ->
    let text =
      try
        let inc = open_in filename in
        let text = Text.read inc in
        close_in inc;
        text
      with exn ->
        Error.error_exn (spf "error reading file %s" filename) exn;
        Text.create ""
    in
    let buf = create filename (Some filename) text local_map in
    Hashtbl.add edt.edt_files filename buf;
    buf
```

⟨*function* Text.read 58e⟩≡ (? 0—1)
```
let read inc =
  create (Utils.read_string inc)
```

Inserting a file

⟨*loading keys* 58f⟩+≡ (25) ◁58a
```
[c_x; NormalMap, Char.code 'i'], Multi_buffers.insert_file;
```

⟨*function* Complex.insert_file 58g⟩≡ (? 0—1)
```
let insert_file frame =
  Select.select_file_from_pwd frame "Insert file: " (fun str ->
    let inc = open_in str in
    Edit.insert_string frame (Utils.read_string inc);
    close_in inc
  )
[@@interactive]
```

⟨*function* `Simple.insert_string` 59a⟩≡ (? 0—1)
```
let insert_string frame str =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.insert text point str;
  Text.fmove text point (String.length str)
```

Saving the buffer

⟨*saving keys* 59b⟩≡ (25) 60c▷
```
[c_x; ControlMap, Char.code 's'], Multi_buffers.save_buffer;
```
Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Complex.save_buffer` 59c⟩≡ (? 0—1)
```
let save_buffer frame =
  let buf = frame.frm_buffer in
  match buf.buf_filename with
  | Some _ -> Ebuffer.save buf
  | None -> write_buffer frame
[@@interactive]
```

⟨`Efuns.buffer` *history fields* 59d⟩+≡ (15a) ◁15e 100b▷
```
mutable buf_last_saved : Text.version;
```

⟨*function* `Ebuffer.save` 59e⟩≡ (? 0—1)
```
let save buf =
  Hooks.exec_named_buf_hooks_with_abort !!save_buffer_hooks buf;

  let filename =
    match buf.buf_filename with
      None -> raise Not_found
    | Some name -> name
  in
  let outc = open_out filename in
  Text.save buf.buf_text outc;
  close_out outc;
  buf.buf_last_saved <- Text.version buf.buf_text;

  let hooks = try Var.get_var buf saved_buffer_hooks with Not_found -> [] in
  Hooks.exec_hooks hooks buf
```

⟨*function* `Text.save` 59f⟩≡ (? 0—1)
```
let save text outc =
  let str = text.text_string in
  let gpos = text.gpoint.pos in
  let gsize = text.gsize in
  output outc str 0 gpos;
  (* skipping the gap *)
  output outc str (gpos + gsize) (text.text_size - gpos - gsize)
```

Save and change buffer hooks

⟨*constant* `Ebuffer.saved_buffer_hooks` 59g⟩≡ (? 0—1)
```
let saved_buffer_hooks = Store.create_abstr "saved_buffer_hooks"
```

⟨*constant* `Complex.buf_mtime` 59h⟩≡ (? 0—1)
```
let buf_mtime = Store.create_float "buf_mtime"
```

⟨*function* Complex.update_time 60a⟩≡ (? 0—1)
```
let update_time buf =
  try
    buf.buf_filename |> Option.iter (fun file ->
      let st = Unix.lstat file in
      if st.Unix.st_kind = Unix.S_REG
      then Var.set_local buf buf_mtime st.Unix.st_mtime;
    )
  with _ -> ()
```

⟨*function* Complex.check_file 60b⟩≡ (? 0—1)
```
let check_file frame =
  try
    let buf = frame.frm_buffer in
    buf.buf_filename |> Option.iter (fun file ->
      let st = Unix.lstat file in
      if st.Unix.st_kind = Unix.S_REG then
        try
          let time = Var.get_local buf buf_mtime in
          Var.set_local buf buf_mtime st.Unix.st_mtime;
          if time <> st.Unix.st_mtime then
            (Select.select_yes_or_no frame
               (Printf.sprintf "%s changed on disk; reload (y/n) ?"
                    buf.buf_name)
                (fun bool ->
                   if bool
                   then reload frame
                   else Frame.status_modified frame true
                )) |> ignore
       with _ -> Var.set_local buf buf_mtime st.Unix.st_mtime
    )
  with _ -> ()
[@@interactive]
```

More saving operations

⟨*saving keys* 60c⟩+≡ (25) ◁59b
```
[c_x; NormalMap,  Char.code 's'], Multi_buffers.save_some_buffers;
[c_x; ControlMap, Char.code 'w'], Multi_buffers.write_buffer;
```

⟨*function* Complex.save_some_buffers 60d⟩≡ (? 0—1)
```
let save_some_buffers frame =
  let buffers = Utils.list_of_hash (Globals.editor()).edt_buffers in
  save_buffers_and_action frame buffers (fun _ -> ())
[@@interactive]
```

⟨*function* Complex.write_buffer 60e⟩≡ (? 0—1)
```
let write_buffer frame =
  let buf = frame.frm_buffer in
  Select.select_file_from_pwd frame "Save file as: " (fun str ->
    Ebuffer.change_name buf str;
    Ebuffer.save buf
  )
[@@interactive]
```

⟨*function* Frame.save_buffer 60f⟩≡ (? 0—1)
```
let save_buffer (frame : t) : unit =
  Ebuffer.save frame.frm_buffer
```

⟨*exception* Ebuffer.BufferAlreadyOpened 61a⟩≡ (? 0—1)

```
exception BufferAlreadyOpened
```

⟨*function* Ebuffer.change_name 61b⟩≡ (? 0—1)

```
let change_name buf filename =
  let edt = Globals.editor() in
  Hashtbl.remove edt.edt_buffers buf.buf_name;
  buf.buf_filename |> Option.iter (fun filename ->
    Hashtbl.remove edt.edt_files filename
  );
  let filename =
    if Filename.is_relative filename
    then Filename.concat edt.edt_dirname filename
    else filename
  in
  if Utils.hashtbl_mem edt.edt_files filename
  then raise BufferAlreadyOpened;
  let filename = Utils.normal_name edt.edt_dirname filename in
  let name = get_unique_name filename in
  Hashtbl.add edt.edt_buffers name buf;
  Hashtbl.add edt.edt_files filename buf;
  buf.buf_filename <- Some filename;
  buf.buf_name <- name
```

⟨*function* Complex.save_buffers_and_action 61c⟩≡ (? 0—1)

```
let rec save_buffers_and_action frame buffers action =
  match buffers with
    [] -> let () = action frame in ()
  | (_,buf) :: buffers ->
      let text = buf.buf_text in
      if buf.buf_last_saved =|= Text.version text  ||
        buf.buf_name.[0] =$= '*'
      then
        save_buffers_and_action frame buffers action
      else
      let map = Keymap.create () in
      let request = Printf.sprintf "Save buffer %s ? (y,n,!,a)" buf.buf_name
      in
      let yes mini_frame =
        Minibuffer.kill mini_frame frame;
        Ebuffer.save buf;
        save_buffers_and_action frame buffers action
      in
      let no mini_frame =
        Minibuffer.kill mini_frame frame;
        save_buffers_and_action frame buffers action; ()
      in
      let action_immediately mini_frame =
        Minibuffer.kill mini_frame frame;
        let () = action mini_frame in ()
      in
      let abort mini_frame =
        Minibuffer.kill mini_frame frame
      in
      Keymap.add_binding map [NormalMap, Char.code 'y'] yes;
      Keymap.add_binding map [NormalMap, Char.code 'Y'] yes;
      Keymap.add_binding map [NormalMap, Char.code 'n'] no;
      Keymap.add_binding map [NormalMap, Char.code 'N'] no;
      Keymap.add_binding map [NormalMap, Char.code '!'] action_immediately;
      Keymap.add_binding map [NormalMap, Char.code 'a'] abort;
```

```
    Keymap.add_binding map [NormalMap, Char.code 'A'] abort;
    Keymap.add_binding map [ControlMap, Char.code 'g'] abort;
    Minibuffer.create frame map request |> ignore
```

Exit

⟨*misc keys* 62a⟩+≡ (25) ◁38a 72d▷
```
  [c_x; ControlMap, Char.code 'c'], Misc_features.exit;
```
Uses `Efuns.mod_ident.ControlMap`.

⟨*function* `Complex.exit_efuns` 62b⟩≡ (? 0—1)
```
  let exit frame =
    let buffers = Utils.list_of_hash (Globals.editor()).edt_buffers in
    let hooks = Var.get_global exit_hooks in
    Hooks.exec_hooks hooks ();
    Multi_buffers.save_buffers_and_action frame buffers (fun _ ->
      raise (Exit.ExitCode 0)
    )
  [@@interactive]
```

6.1.2 Edit menu

The edit menu mirrors standard clipboard operations (cut, paste, undo), again mapping each entry to a named action string that resolves through the global `actions` hash table.

⟨*constant* `Top_window.edit_menu` 62c⟩≡ (? 0—1)
```
  let edit_menu = define_option ["edit_menu"] "" (list_option string2_option) []
```

⟨`Config` *edit menu setup* 62d⟩≡ (57a)
```
  if !!Top_window.edit_menu =*= [] then begin
      Top_window.edit_menu =:= [
        "Cut",    "kill_region";
        "Paste",  "insert_killed";

        "Undo",    "undo";
        "", "";
        ⟨edit_menu entries 122c⟩
      ];
    end;
```

6.1.3 Buffers menu

The buffers menu is dynamically generated (seen later with the buffer-management code), so only a placeholder appears here.

6.1.4 Help menu

Unlike the other menus, the help menu stores direct action closures rather than action-name strings, and uses a plain `ref` rather than an option. Its entries open special buffers (`*bindings*`, `*help*`) via `Frame.change_bufferX`, which we will see later.

⟨*constant* `Top_window.help_menu` 62e⟩≡ (? 0—1)
```
  let help_menu = ref ([| |]: (string * action) array)
```

⟨Config *help menu setup* 63a⟩≡ (57a)
```
Top_window.help_menu := [|
  "Key Bindings", (fun frame ->
    Frame.change_buffer frame.caps frame.frm_window "*bindings*"
  );
  "About Efuns", (fun frame ->
    Frame.change_buffer frame.caps frame.frm_window "*help*"
  );
  "Changes", (fun _frame ->
    (*
      let _ = Frame.load_file frame.frm_window (
          Version.efuns_lib ^"/Changes") in ()
    *)
    failwith "Std_xxx.menu changes: TODO"
  );
|];
```
Uses Efuns.frame.frm_window 16a.

6.2 Sub-windows

This section deals with splitting the editing area into panes and rendering their content. Each pane is a `Window.t`/`Frame.t` pair: the window provides geometry, the frame provides the view of a buffer. The key function is `Frame.displayX`, which decides what lines are visible and repaints only those that have changed.

6.2.1 Frames and windows creation

We already saw `Frame.create_without_topX` in the Core Data Structures chapter. Here we see `Frame.installX`, which binds a frame to a window: it kills any previous frame in that window, copies the window's geometry into the frame's fields, and marks it for redraw. This is how `C-x b` (switch buffer) or window splitting takes effect.

⟨*function* Frame.install 63b⟩≡ (? 0—1)
```
let install window frame =
  ⟨Frame.install() sanity check frame is not a minibuffer 78c⟩

  window |> Window.iter (fun f ->
     (* TODO? != or <> ? *)
     if (Eq.phys_not_equal f frame)
     then kill f
  );
  window.win_down <- WFrame frame;

  frame.frm_xpos <- window.win_xpos;
  frame.frm_ypos <- window.win_ypos;
  frame.frm_width <- window.win_width;
  frame.frm_height <- window.win_height;
  frame.frm_window <- window;

  ⟨Frame.install() adjust frm_cutline 66b⟩
  ⟨Frame.install() set frm_table 66f⟩
  frame.frm_redraw <- true
```

6.2.2 Windows display

We saw in the startup chapter how `Top_window.update_displayX` iterates over all windows via `Window.iter`. Here we focus on what happens inside each individual frame.

6.2.3 Frame display

`Frame.displayX` is the heart of the redisplay engine. It is called once per frame by `Top_window.update_displayX` and decides whether anything needs repainting. A redraw is triggered if the cursor has moved outside the visible region (`frm_start`..`frm_end`), if the buffer has been modified since the last paint, or if `frm_redraw` has been set explicitly (e.g., by a window resize or split). When a redraw is needed, the function first adjusts offsets so that `frm_start` is at a beginning-of-line, calls `update_tableX` to fill `frm_table` with the lines that should appear on screen, and then iterates over the table, painting only lines whose box list has changed since last time.

`Frame.display()`

⟨*function* `Frame.display` 64a⟩≡ (? 0—1)
```
let display top_window frame =
  let (buf, text, point) = buf_text_point frame in

  let width = frame.frm_width in
  let height = frame.frm_height - frame.frm_has_status_line in

  let graphic = Efuns.backend top_window in

  ⟨Frame.display() if buf sync goto end of text 137c⟩
  if
    ⟨Frame.display() conditions for redraw 64d⟩
  then begin
    ⟨Frame.display() redraw 65f⟩
  end;
  ⟨Frame.display() draw status line or minibuffer 64c⟩
[@@profiling]
```

Frame decorations display

⟨`Efuns.frame` *decoration fields* 64b⟩≡ (16a)
```
(* 0 for minibuffer, 1 for normal frame *)
mutable frm_has_status_line : int;
(* Some for minibuffer, None for normal frame *)
mutable frm_mini_buffer : string option;
```
Uses `Efuns.frm_line` 74b.

⟨`Frame.display()` *draw status line or minibuffer* 64c⟩≡ (64a)
```
match frame.frm_mini_buffer with
| None ->
    ⟨Frame.display() draw status line 76d⟩
| Some request ->
    ⟨Frame.display() draw minibuffer request string 79f⟩
```

Conditions for redraw

⟨`Frame.display()` *conditions for redraw* 64d⟩≡ (64a)
```
⟨Frame.display() conditions for redraw, point outside frame 65a⟩
⟨Frame.display() conditions for redraw, buffer modified 65c⟩
⟨Frame.display() conditions for redraw, forced redraw 65e⟩
```

⟨`Efuns.frame` *current position fields* 64e⟩+≡ (16a) ◁16c
```
(* first point of the first buffer-line on screen *)
mutable frm_start : Text.point;
(* last point on screen, -1 if modified *)
mutable frm_end : Text.point;
```

⟨Frame.display() *conditions for redraw, point outside frame* 65a⟩≡ (64d)
```
(point < frame.frm_start) ||
(point > frame.frm_end)  ||
```

⟨Efuns.frame *history fields* 65b⟩≡ (16a)
```
mutable frm_last_text_updated : int;
mutable frm_last_buf_updated : int;
```

⟨Frame.display() *conditions for redraw, buffer modified* 65c⟩≡ (64d)
```
(version text <> frame.frm_last_text_updated) ||
(buf.buf_modified <> frame.frm_last_buf_updated) ||
```

⟨Efuns.frame *other fields* 65d⟩≡ (16a) 66a▷
```
mutable frm_redraw : bool;
```

⟨Frame.display() *conditions for redraw, forced redraw* 65e⟩≡ (64d)
```
frame.frm_redraw
```

Redraw

⟨Frame.display() *redraw* 65f⟩≡ (64a)
```
if !Globals.debug_graphics
then Logs.debug (fun m -> m "redraw");
⟨Frame.display() redraw, possibly update frm_y_offset 71e⟩
⟨Frame.display() redraw, possibly update frm_x_offset 72c⟩
(* invariant: now frm_start is at a bol *)
update_table frame;
(* invariant: now frm_end has been correctly set *)

if (point > frame.frm_end) || (point < start) then begin
    ⟨Frame.display() redraw, if frm_force_start 73b⟩
    else begin
      (* center around point *)
      Text.goto_point text start point;
      ⟨Frame.display() redraw, update frm_y_offset again 71f⟩
      (* invariant: now frm_start is at a bol again *)
      update_table frame;
   end
end;

frame.frm_force_start <- false;

frame.frm_last_text_updated <- version text;
frame.frm_last_buf_updated <- buf.buf_modified;

for y = 0 to height - 1 do
  ⟨Frame.display() redraw, draw line y if line changed 67c⟩
done;
frame.frm_redraw <- false
```

6.2.4 Line display, cutting long lines

There are four coordinate spaces to keep straight: (1) positions in `text_string`, (2) line/column in `text_newlines`, (3) positions in `repr_string` (after expanding tabs and control characters), and (4) x/y in the frame (after wrapping long lines at `frm_cutline`). The `frm_table` bridges spaces (3) and (4): each `frm_line` entry holds a slice

of a `Text.line`'s boxes, possibly starting partway through a box when a long line wraps. A single text line can thus generate multiple `frm_line` entries, each terminated by a "/" overflow marker.

⟨`Efuns.frame` *other fields* 66a⟩+≡ (16a) ◁65d 66c▷
```
(* where do we put a \ for overflowing lines *)
mutable frm_cutline : int; (* max_int for no, else length *)
```

⟨`Frame.install()` *adjust* `frm_cutline` 66b⟩≡ (63b)
```
if frame.frm_cutline < max_int
then frame.frm_cutline <- window.win_width - 1;
```

The frame table

⟨`Efuns.frame` *other fields* 66c⟩+≡ (16a) ◁66a 69b▷
```
mutable frm_table : frm_line array;
```

⟨*type* `Efuns.frm_line` 66d⟩≡ (? 0—1)
```
and frm_line =
  {
    mutable frm_text_line : Text.line;

    (* sorted normally, head = first box in line *)
    mutable frmline_boxes : Text.box list;
    mutable first_box_extra_offset : int;

    ⟨Efuns.frm_line other fields 66e⟩
  }
```
Uses `Efuns.frame`.

⟨`Efuns.frm_line` *other fields* 66e⟩≡ (66d) 67d▷
```
mutable lineidx_in_text : int;
```

⟨`Frame.install()` *set* `frm_table` 66f⟩≡ (63b)
```
frame.frm_table <- (Array.init window.win_height (fun _i ->
   {
     frm_text_line = Text.dummy_line;
     frmline_boxes = [];

     lineidx_in_text = 0;
     first_box_extra_offset = 0;

     repr_prev_offset = 0;
     prev_frmline_boxes = [];
   }
));
```

`Frame.update_table()`

⟨*function* `Frame.update_table` 66g⟩≡ (? 0—1)
```
let update_table frame =
  let (buf, text, _) = buf_text_point frame in

  let start = frame.frm_start in
  let height = frame.frm_height - frame.frm_has_status_line in

  let current_n = ref (Text.point_line text start) in
  let current_line = ref (Ebuffer.compute_representation buf !current_n) in

  ⟨Frame.update_table() adjust current line when frm_y_offset negative 72b⟩
```

```
⟨Frame.update_table() adjust current line when frm_y_offset positive 72a⟩

(* update frame.frm_start *)
Text.goto_line text start !current_n;

(* update frame representation *)
⟨function Frame.update_table.iter_line 67a⟩
⟨function Frame.update_table.iter_repr 67b⟩

iter_line (- frame.frm_y_offset) !current_n !current_line
[@@profiling]
```

⟨*function* `Frame.update_table.iter_line` 67a⟩≡ (66g)

```
let rec iter_line y n line =
  if y < height then begin
    let reprs = List.rev line.boxes in
    if y >= 0 then begin
        let frm_line = frame.frm_table.(y) in
        frm_line.frm_text_line <- line;
        frm_line.frmline_boxes <- reprs;
        frm_line.lineidx_in_text <- n;
        frm_line.first_box_extra_offset <- 0;
    end;
    iter_repr frame.frm_cutline (y+1) n line reprs
  end
  else Text.goto_line text frame.frm_end (n-1)
```

⟨*function* `Frame.update_table.iter_repr` 67b⟩≡ (66g)

```
and iter_repr xcutline y n line boxes =
  ⟨Frame.update_table.iter_repr() if line too big 68b⟩
  else
    let line = Ebuffer.compute_representation buf (n+1) in
    iter_line y (n+1) line
in
```

`Frame.display_line()`

⟨`Frame.display()` *redraw, draw line y if line changed* 67c⟩≡ (65f)

```
let frm_line = frame.frm_table.(y) in

if not ((Eq.phys_equal frm_line.prev_frmline_boxes frm_line.frmline_boxes) &&
        (Eq.phys_equal frm_line.repr_prev_offset frm_line.first_box_extra_offset))
   || frame.frm_redraw
then
  begin
    frm_line.prev_frmline_boxes <- frm_line.frmline_boxes;
    frm_line.repr_prev_offset <- frm_line.first_box_extra_offset;

    display_line graphic frame
        (Bytes.to_string frm_line.frm_text_line.repr_string)
        y;
  end;
```

⟨`Efuns.frm_line` *other fields* 67d⟩+≡ (66d) ◁66e

```
(* previous values, so can check if the line changed *)
mutable repr_prev_offset : int;
mutable prev_frmline_boxes : Text.box list;
```

Uses `Xdraw.graphics_backend`.

⟨*function* `Frame.display_line` 68a⟩≡ (? 0—1)

```
let display_line graphic frame repr_string y =
  let frm_line = frame.frm_table.(y) in

  let rec iter x offset boxes =
    if x < frame.frm_width then
      match boxes with
      | [] ->
          graphic.Xdraw.clear_eol (x+frame.frm_xpos) (y+frame.frm_ypos)
            (frame.frm_width - x)
      | box :: tail ->
          let len = min (frame.frm_width-x) (box.box_size - offset) in
          graphic.Xdraw.draw_string (x+frame.frm_xpos) (y+frame.frm_ypos)
            repr_string (box.box_pos_repr+offset) len
            box.box_attr;
          iter (x+len) 0 tail
    else
      ⟨Frame.display_line() in iter, line overflow frm_width 68c⟩
  in
  iter 0 (frm_line.first_box_extra_offset + frame.frm_x_offset) frm_line.frmline_boxes
```

Overflowing lines

⟨`Frame.update_table.iter_repr()` *if line too big* 68b⟩≡ (67b)

```
if line.repr_len > xcutline then
  match boxes with
  | box :: tail ->
      if box.box_pos_repr <= xcutline &&
         box.box_pos_repr + box.box_size > xcutline
      then
        if y =|= height
        then Text.goto_line text frame.frm_end n
        else begin
            if y>= 0 then begin
              let frm_line = frame.frm_table.(y) in
              frm_line.frm_text_line <- line;
              frm_line.lineidx_in_text <- n;
              frm_line.first_box_extra_offset <- xcutline - box.box_pos_repr;
              frm_line.frmline_boxes <- boxes;
            end;
            iter_repr (xcutline+frame.frm_cutline) (y+1) n line boxes
        end
      else iter_repr xcutline y n line tail
  | [] ->
      let line = Ebuffer.compute_representation buf (n + 1) in
      iter_line (y+1) (n+1) line
```

⟨`Frame.display_line()` *in iter, line overflow* `frm_width` 68c⟩≡ (68a)

```
graphic.Xdraw.draw_string
   (frame.frm_width+frame.frm_xpos-1) (y+frame.frm_ypos)
   "/" 0 1 Text.direct_attr
```

6.2.5 Cursor display

Cursor display is the reverse of line display: given the buffer point, find its x/y position in the frame. This requires mapping through all four coordinate spaces (text position → line/column → `repr_string` position → frame x/y with overflow wrapping). The cursor is drawn by saving the character and attribute under it

(`frm_cursor`, `frm_cursor_attr`), then redrawing that character in `inverse_attr`. The "cursor off" phase restores the original. A secondary cursor (`top_second_cursor`) supports features like incremental search, where the minibuffer is active but the search result cursor should be visible in the main frame.

Cursor on

⟨*function* `Top_window.cursor_on` 69a⟩≡ (? 0—1)
```
let cursor_on top_window =
  let frame = top_window.top_active_frame in
  set_cursor_on top_window frame;
  ⟨Top_window.cursor_on() set top window name 73e⟩
  ⟨Top_window.cursor_on() display secondary cursor 92b⟩
```

⟨`Efuns.frame` *other fields* 69b⟩+≡ (16a) ◁66c 71d▷
```
(* frame coordinates, so x is between 0 and frm_width - 1 *)
mutable frm_cursor_x : int;
(* frame coordinates, so y is between 0 and frm_height - 1 *)
mutable frm_cursor_y : int;

mutable frm_cursor : bytes;
mutable frm_cursor_attr : Text.attribute;
```
Uses `Efuns.action`.

⟨*function* `Top_window.set_cursor_on` 69c⟩≡ (? 0—1)
```
let set_cursor_on top_window frame =
  Frame.set_cursor frame;

  if (Bytes.get frame.frm_cursor 0) <> '\000' then
    let graphic = Efuns.backend top_window in
    graphic.Xdraw.draw_string
      (frame.frm_xpos + frame.frm_cursor_x-frame.frm_x_offset)
      (frame.frm_ypos + frame.frm_cursor_y)
      (Bytes.to_string frame.frm_cursor) 0 1 Text.inverse_attr
```

⟨*function* `Frame.set_cursor` 69d⟩≡ (? 0—1)
```
let set_cursor frame =
  let (buf, text, point) = buf_text_point frame in

  match point_to_xy_opt frame point with
  | None ->
      (* insert cursor is not on frame *)
      Bytes.set frame.frm_cursor 0 '\000'
  | Some (x, y) ->
      frame.frm_cursor_x <- x;
      frame.frm_cursor_y <- y;
      let col = point_col text point in
      let line_ = point_line text point in
      let line = Ebuffer.compute_representation buf line_ in

      let rec iter boxes =
        match boxes with
        | [] ->
            Bytes.set frame.frm_cursor 0 ' '
        | box :: tail ->
            if box.box_col <= col && box.box_col + box.box_len > col
            then begin
              let pos_repr =
                box.box_pos_repr + box.box_charsize *
                (col - box.box_col)
```

```
              in
              Bytes.set frame.frm_cursor 0
                  (Bytes.get line.repr_string pos_repr);
              frame.frm_cursor_attr <- box.box_attr;
            end else
              iter tail
      in
      let frm_line = frame.frm_table.(frame.frm_cursor_y) in
      iter frm_line.frmline_boxes
```

Point coordinates vs xy screen coordinates

⟨*function* Frame.point_to_xy_opt 70a⟩≡ (? 0—1)

```
let point_to_xy_opt frame point =
  let (buf, text, _) = buf_text_point frame in

  let x_nocut = point_to_x_when_no_cutline buf point in
  let line = Ebuffer.compute_representation buf (point_line text point) in

  let xref = ref 0 in
  let yref = ref 0 in

  try
    for i = 0 to frame.frm_height - 1 do
      let line_repr = frame.frm_table.(i) in
      if Eq.phys_equal line_repr.frm_text_line line then
        let x,y =
          if x_nocut =|= 0
          then 0,i
          else
            ⟨Frame.point_to_xy_opt() x y value handling overflow lines 70b⟩
        in
        xref := x;
        yref := y;
        raise Exit
    done;
    None
  with Exit ->
    (* claude: when a long line wraps past the bottom of the visible frame,
     * the computed y can exceed frm_height - 1. Return None (cursor not
     * visible) to avoid an out-of-bounds access in frm_table by set_cursor. *)
    if !yref >= frame.frm_height
    then None
    else Some (!xref, !yref)
```

⟨Frame.point_to_xy_opt() *x y value handling overflow lines* 70b⟩≡ (70a)

```
((x_nocut-1) mod frame.frm_cutline) + 1, i + (x_nocut-1) / frame.frm_cutline
```

⟨*function* Frame.point_to_cursor 70c⟩≡ (? 0—1)

```
let point_to_x_when_no_cutline buf point =
  let text = buf.buf_text in
  let line = Ebuffer.compute_representation buf (Text.point_line text point) in
  let xpos = Text.point_col text point in
  let rec iter boxes =
    match boxes with
    | [] -> 0
    | box :: tail ->
        if xpos < box.box_col
        then iter tail
```

```
        else box.box_pos_repr + box.box_charsize *  (xpos - box.box_col)
  in
  iter line.Text.boxes
```

Cursor off

⟨*function* `Top_window.clean_display` 71a⟩≡ (? 0—1)
```
let clean_display () =
  (Globals.editor()).top_windows |> List.iter (fun top_window ->
     cursor_off top_window;
     ⟨clean_display() sanity check second cursor and minibuffer 92d⟩
   )
```

⟨*function* `Top_window.cursor_off` 71b⟩≡ (? 0—1)
```
let cursor_off top_window =
  let frame = top_window.top_active_frame in
  set_cursor_off top_window frame;
  ⟨Top_window.cursor_off() hide secondary cursor 92c⟩
```

⟨*function* `Top_window.set_cursor_off` 71c⟩≡ (? 0—1)
```
let set_cursor_off top_window frame =
  if (Bytes.get frame.frm_cursor 0) <> '\000' then
    let graphic = Efuns.backend top_window in
    graphic.Xdraw.draw_string
      (frame.frm_xpos + frame.frm_cursor_x)
      (frame.frm_ypos + frame.frm_cursor_y)
      (Bytes.to_string frame.frm_cursor) 0 1 frame.frm_cursor_attr
```

6.2.6 Frame offsets

The frame offsets handle the gap between text lines and screen lines. Because a single text line may wrap into multiple screen lines (due to `frm_cutline`), simply setting `frm_start` to a text line is not enough to pin the viewport. `frm_y_offset` tracks how many screen lines to skip (positive) or backtrack (negative) from `frm_start`, while `frm_x_offset` provides horizontal scrolling when `frm_cutline` is disabled (`max_int`). The `recenter` command (`C-l`) resets these offsets to center the cursor vertically.

⟨`Efuns.frame` *other fields* 71d⟩+≡ (16a) ◁69b 73a▷
```
(* offset(+/-) of screen-lines after frm_start *)
mutable frm_y_offset : int;
(* ?? *)
mutable frm_x_offset : int;
```

Y offset

⟨`Frame.display()` *redraw, possibly update* `frm_y_offset` 71e⟩≡ (65f)
```
let start = frame.frm_start in
let start_c = point_to_x_when_no_cutline buf start in
if start_c > 0 then begin
  frame.frm_y_offset <- frame.frm_y_offset - start_c / frame.frm_cutline;
  Text.bmove text start start_c
end;
```

⟨`Frame.display()` *redraw, update* `frm_y_offset` *again* 71f⟩≡ (65f)
```
frame.frm_y_offset <- - height / 2;
let start_c = point_to_x_when_no_cutline buf start in
if start_c > 0 then begin
  frame.frm_y_offset <- frame.frm_y_offset - start_c / frame.frm_cutline;
  Text.bmove text start start_c
end;
```

⟨Frame.update_table() *adjust current line when* frm_y_offset *positive* 72a⟩≡ (66g)

```
(* assert current_line is the first line *)

while frame.frm_y_offset > !current_line.repr_len / frame.frm_cutline
    && !current_n < nbr_lines text
do
  frame.frm_y_offset <- frame.frm_y_offset -
    (!current_line.repr_len / frame.frm_cutline) - 1;
  current_n := !current_n + 1;
  current_line := Ebuffer.compute_representation buf !current_n;
done;

if !current_n =|= nbr_lines text &&
   frame.frm_y_offset > !current_line.repr_len / frame.frm_cutline
then frame.frm_y_offset <- !current_line.repr_len / frame.frm_cutline;
```

⟨Frame.update_table() *adjust current line when* frm_y_offset *negative* 72b⟩≡ (66g)

```
(* assert frame.frm_y_offset >= 0 *)

while frame.frm_y_offset < 0 && !current_n > 0 do
  current_n := !current_n - 1;
  current_line := Ebuffer.compute_representation buf !current_n;
  let lines = !current_line.repr_len / frame.frm_cutline in
  frame.frm_y_offset <- frame.frm_y_offset + lines + 1;
done;
if !current_n =|= 0 && frame.frm_y_offset <0
then frame.frm_y_offset <- 0;
```

X offset

⟨Frame.display() *redraw, possibly update* frm_x_offset 72c⟩≡ (65f)

```
let point_c = point_to_x_when_no_cutline buf point in
if point_c < frame.frm_x_offset then begin
    frame.frm_x_offset <- max (point_c - width / 2) 0;
    frame.frm_redraw <- true;
end else if frame.frm_cutline =|= max_int &&
            (point_c mod frame.frm_cutline >= frame.frm_x_offset + width - 3)
  then begin
    frame.frm_x_offset <- point_c - (width / 2);
    frame.frm_redraw <- true;
end;
```

Recenter

⟨*misc keys* 72d⟩+≡ (25) ◁62a 84a▷

```
[ControlMap, Char.code 'l'], Scroll.recenter;
```

Uses Efuns.mod_ident.NormalMap.

⟨*function* Simple.recenter 72e⟩≡ (? 0—1)

```
let recenter frame =
  let (_, text, point) = Frame.buf_text_point frame in

  frame.frm_force_start <- true;
  frame.frm_redraw <- true;
  Text.goto_point text frame.frm_start point;
  frame.frm_y_offset <- - frame.frm_height/2
[@@interactive]
```

⟨Efuns.frame *other fields* 73a⟩+≡ (16a) ◁71d 79d▷
```
  mutable frm_force_start : bool;
```

⟨Frame.display() *redraw, if* frm_force_start 73b⟩≡ (65f)
```
  if frame.frm_force_start then begin
    let coord = cursor_to_coord frame frame.frm_cursor_x frame.frm_cursor_y in
    Text.goto_line text frame.frm_point coord.Text.c_line;
    Text.fmove text frame.frm_point coord.Text.c_col
  end
```

⟨*function* Top_window.scroll_to_frame 73c⟩≡ (? 0—1)
```
  (*
  let scroll_to_frame ady top_window =
    let frame = top_window.top_active_frame in
    let (buf, text, _) = Frame.buf_text_point frame in
    let pos_start = Text.get_position text frame.frm_start in
    let _size = Text.size text in
    UCommon.pr2 "ady#get_pos size";
    let y = 1
  (*    ady#get_pos size  *)
    in
    if abs (y - pos_start) > 80 then
    (* the position has changed *)
      begin
        frame.frm_force_start <- true;
        frame.frm_redraw <- true;
        Text.set_position text frame.frm_start y
      end
  *)
```

6.2.7 Top windows name

The OS-level window title is updated during `cursor_on` to reflect the active buffer's name, but only when it has actually changed (avoiding redundant backend calls).

⟨Efuns.top_window *other fields* 73d⟩+≡ (19b) ◁19e 77a▷
```
  mutable top_name : string;
```
Uses Efuns.window_down 18b.

⟨Top_window.cursor_on() *set top window name* 73e⟩≡ (69a)
```
  let name = frame.frm_buffer.buf_name in
  if name <> top_window.top_name then begin
    let graphic = Efuns.backend top_window in
    graphic.Xdraw.update_window_title name;
    top_window.top_name <- name
  end;
```

6.3 Status Line

The status line is the inverse-video bar at the bottom of each frame, showing the buffer name, modification flag (** or --), cursor position, scroll percentage, and active modes. It is implemented as a fixed-width byte string (`status_string`) with a format table that maps each `status_info` variant to a (position, max_length) pair. The `status_print` function blits updated values into the string at their designated positions, and the display code draws the whole string in one `draw_string` call with `inverse_attr`.

6.3.1 Status line creation

Each frame carries its own `status` record, pre-filled at frame creation time. The `status_string` is a 256-byte buffer initialised with dashes, into which the editor name ("Efuns:") is blitted at a fixed offset. The `stat_modified` flag tracks whether `status_string` has changed since the last repaint, and `status_modified` is the double-starred "buffer modified" indicator.

⟨`Efuns.frame` *status field* 74a⟩≡ (16a)
```
mutable frm_status : status;
```

⟨*type* `Efuns.status` 74b⟩≡ (? 0—1)
```
and status =
  {
    mutable stat_name : string;
    mutable stat_file : string;
    mutable stat_line : int;
    mutable stat_col : int;

    mutable status_modified : bool;
    mutable stat_modified : bool;
    ⟨Efuns.status other fields 75a⟩
  }
```
Uses `Text.line`.

⟨`Frame.create_without_top()` *let status* 74c⟩≡ (17b)
```
let status = {
    stat_name = "";
    stat_file = "";
    stat_line = -1;
    stat_col = -1;
    status_modified = true;
    stat_modified = (buf.buf_last_saved =|= Text.version buf.buf_text);

    stat_modes = [];
    stat_mode = dummy_mode;

    status_format = !status_format;

    status_string = Bytes.make 256 '-';
  } in
String.blit editname 0 status.status_string 5 (String.length editname);
```

⟨*constant* `Frame.dummy_mode` 74d⟩≡ (? 0—1)
```
let dummy_mode = Ebuffer.new_major_mode "" None
```

⟨*constant* `Frame.editname` 74e⟩≡ (? 0—1)
```
let editname = "Efuns:"
```

⟨`Frame.create_without_top()` *adjust status of frame* 74f⟩≡ (17b) 128d▷
```
status_name frame buf.buf_name;
```

⟨*function* `Frame.status_name` 74g⟩≡ (? 0—1)
```
let status_name frame name =
  let status = frame.frm_status in
  if status.stat_name <> name then begin
    status.stat_name <- name;
    status_print status name StatName
  end
```

6.3.2 Status line format

The format is a list of (`status_info`, (position, max_length)) pairs, giving each field a fixed column range in `status_string`. This table-driven layout makes it easy to rearrange the status bar without touching the rendering code.

⟨`Efuns.status` *other fields* 75a⟩≡ (74b) 75e▷
```
(* the pair of int is a:   pos * max_len *)
mutable status_format : (status_info * (int * int)) list;
```
Uses `Text.box`.

⟨*type* `Efuns.status_info` 75b⟩≡ (? 0—1)
```
and status_info =
| StatName
| StatFile
| StatLine
| StatCol
| StatModified
| StatMode
```
Uses `Efuns.status_info`.

⟨*constant* `Frame.status_format` 75c⟩≡ (? 0—1)
```
let status_format = ref [
    StatModified, (1, 2);
    StatName, (5, 20);
    StatMode, (30,30);
    StatLine, (65, 5);
    StatCol, (70, 5);
    StatFile, (35,15);
  ]
```

6.3.3 `Frame.status_print()`

`status_printX` is the central update function: it looks up the stat type in the format table, blits the new value into `status_string` at the right position, and pads the rest with spaces. If the stat type is not in the format (e.g., a custom layout), the `Not_found` exception is silently caught.

⟨*function* `Frame.status_print` 75d⟩≡ (? 0—1)
```
let status_print status str stat_type =
  status.status_modified <- true;
  try
    let (pos,maxlen) = List.assoc stat_type status.status_format in
    let len = min (String.length str) maxlen in
    Bytes.blit_string str 0 status.status_string pos len;
    Bytes.fill status.status_string (pos + len) (maxlen - len) ' '
  with Not_found -> ()
```

⟨`Efuns.status` *other fields* 75e⟩+≡ (74b) ◁75a 127a▷
```
(* the string! --- ... --- *)
mutable status_string : bytes;
```

6.3.4 `Frame.status_xxx()`

Each `status_xxx` helper updates one field of the status line. They all follow the same pattern: compare the new value against the cached one, and call `status_printX` only when it has changed, avoiding redundant byte

blits on every redisplay cycle.

⟨*function* `Frame.status_modified` 76a⟩≡ (? 0—1)
```
let status_modified frame modified =
  let status = frame.frm_status in
  if status.stat_modified <> modified then begin
    status_print status (if modified then "**" else "--") StatModified;
    status.stat_modified <- modified
  end
```

⟨*function* `Frame.status_line` 76b⟩≡ (? 0—1)
```
let status_line frame line =
  let status = frame.frm_status in
  if status.stat_line <> line then begin
    status.stat_line <- line;
    status_print status (Printf.sprintf "L%d" (line+1)) StatLine
  end
```

⟨*function* `Frame.status_col` 76c⟩≡ (? 0—1)
```
let status_col frame col =
  let status = frame.frm_status in
  if status.stat_col <> col then begin
    status.stat_col <- col;
    status_print status (Printf.sprintf "C%d" (col+1)) StatCol
  end
```

6.3.5 Status line display

The commented-out code shows the original approach: update each `status_xxx` field, then draw `status_string` if anything changed. The active code below takes a simpler path—it formats the entire status line as one `spf` string each time, including the `Top`/`Bot`/percentage indicator and the major+minor mode names, then draws it directly with `inverse_attr`.

⟨`Frame.display()` *draw status line* 76d⟩≡ (64c)
```
(*
let status = frame.frm_status in

status_modified frame (version text <> buf.buf_last_saved);
status_line frame  (point_line text frame.frm_point);
status_col  frame  (point_col  text frame.frm_point);
status_name frame buf.buf_name;
status_major_mode frame;

if status.status_modified
then
  graphic.Xdraw.draw_string
    frame.frm_xpos (frame.frm_ypos + frame.frm_height - 1)
    status.status_string
    0 width Text.inverse_attr
*)
let str = spf "-%s- %s  %s L%-6d (%s) --"
  (if (version text <> buf.buf_last_saved) then "**" else "--")
  (buf.buf_name ^
    String.make (max 1 ((width / 3) - String.length buf.buf_name)) ' ')
  (match () with
  | _ when Text.get_position text point =|= 0 -> "Top"
  | _ when Text.get_position text point =|= Text.size text -> "Bot"
  | _ ->
     let nblines = Text.nbr_lines text in
```

```
      if nblines =|= 0
      then " 0%%"
      else spf "%2d%%" (Text.point_line text point * 100 / nblines)
    )
    (Text.point_line text point + 1)
    (let xs = buf.buf_minor_modes |> List.map (fun m -> m.min_name) in
     let xs = buf.buf_major_mode.maj_name :: xs in
     xs |> String.concat " "
    )
  in
  let minuses = String.make 256 '-' in
  let str = str ^ minuses in
  graphic.Xdraw.draw_string
    frame.frm_xpos (frame.frm_ypos + frame.frm_height - 1)
    str
    0 width Text.inverse_attr
```

6.4 Minibuffer

The minibuffer is Emacs's solution for prompting the user without opening a dialog box: a one-line editing area at the very bottom of the window. In efuns, it is implemented as a real buffer/frame/window stack —the same machinery used for editing files—which means tab completion, history, and even recursive minibuffers come almost for free. Each call to `Minibuffer.createX` pushes a new frame onto `top_mini_buffers`; `C-g` or pressing Return pops it via `Minibuffer.killX`, restoring the previous active frame. The request string (e.g., "Find file:") is drawn directly at the left of the bottom line, and the minibuffer's editable area occupies the remaining columns to its right.

6.4.1 Minubuffer creation

Each minibuffer invocation creates a fresh `Text.t`, wraps it in a buffer with its own local keymap, and opens a tiny one-line window at the bottom of the screen. The `top_mini_buffers` list acts as a stack, so recursive prompts (e.g., a save confirmation during exit) nest correctly. `C-g` is always bound to `Minibuffer.killX` as an escape hatch.

⟨`Efuns.top_window` *other fields* 77a⟩+≡ (19b) ◁73d 92a▷

```
  mutable top_mini_buffers : frame list;
```

⟨*function* `Minibuffer.create` 77b⟩≡ (? 0—1)

```
  let create (frame : Frame.t) (local_map : Keymap.t) (request : string) : Frame.t =
    let window = frame.frm_window in
    let top_window = Window.top window in

    let qlen = String.length request in
    let request =
       if qlen < 50
       then request
       else String.sub request 0 47 ^ "..."
    in

    let mini_text =
      Text.create "" in
    let mini_buf =
      buf_create mini_text local_map in
    let mini_window =
      Window.create true (*mini*) (TopWindow top_window)
        qlen (top_window.top_height - 1) (top_window.top_width - qlen) 1 in
```

```
    let mini_frame =
      Frame.create frame.caps mini_window (Some request)(*mini*) mini_buf in

    mini_frame.frm_cutline <- max_int;
    mini_frame.frm_has_status_line <- 0;

    top_window.top_mini_buffers <- mini_frame :: top_window.top_mini_buffers;

    Keymap.add_binding local_map [ControlMap, Char.code 'g']
      (fun mini_frame -> kill mini_frame frame);
    mini_frame
```

⟨Efuns.window *other fields* 78a⟩+≡ (18b) ◁18e
```
  mutable win_mini : bool;
```

⟨*function* Minibuffer.buf_create 78b⟩≡ (? 0—1)
```
  let buf_create text local_map =
    {
      buf_text = text;

      buf_name = "*Minibuffer*";
      buf_filename = None; (* no connected file! *)

      buf_point = Text.new_point text;
      buf_start = Text.new_point text;

      buf_last_saved = Text.version text;
      buf_modified = 0;


      buf_map = local_map;
      buf_map_partial = true;
      buf_charreprs = charreprs;
      buf_syntax_table = Ebuffer.default_syntax_table;
      buf_vars = Store.new_store ();
      buf_major_mode = Ebuffer.fondamental__mode;
      buf_minor_modes = [];

      buf_sync = false;
      buf_mark = None;
      buf_shared = 0;
      buf_finalizers = [];
      buf_history_pos = [||];
    }
```

⟨Frame.install() *sanity check frame is not a minibuffer* 78c⟩≡ (63b)
```
  (* TODO: dunno why but can't use =:= for bool *)
  if window.win_mini =*= (frame.frm_mini_buffer =*= None) then begin
    kill frame;
    failwith "Cannot install in minibuffer"
  end;
```

6.4.2 Minubuffer kill

Killing a minibuffer pops the stack, clears any secondary cursor left by isearch, and restores the previous active frame. If that frame was killed while the minibuffer was active (e.g., its window was deleted), `Frame.unkillX`

re-installs it, duplicating its points so it can resume safely.

⟨*function* `Minibuffer.kill` 79a⟩≡ (? 0—1)
```
let kill mini_frame old_frame =
  let window = mini_frame.frm_window in
  let top_window = Window.top window in
  Top_window.clear_message top_window;
  top_window.top_mini_buffers <- List.tl top_window.top_mini_buffers;
  (* claude: clear top_second_cursor to avoid stale references.
   * Features like isearch set top_second_cursor before creating a
   * minibuffer, but the C-g cancel path calls kill without clearing it,
   * causing "weird, second cursor" errors on the next display cycle. *)
  top_window.top_second_cursor <- None;
  if old_frame.frm_killed
  then Frame.unkill window old_frame;
  top_window.top_active_frame <- old_frame;
  Frame.kill mini_frame
```

⟨*exception* `Frame.BufferKilled` 79b⟩≡ (? 0—1)
```
exception BufferKilled
```

⟨*function* `Frame.unkill` 79c⟩≡ (? 0—1)
```
let unkill window frame =
  let (buf, text, _) = buf_text_point frame in
  if buf.buf_shared < 0
  then raise BufferKilled;
  install window frame;
  frame.frm_start <- Text.dup_point text buf.buf_start;
  frame.frm_end <- Text.dup_point text buf.buf_start;
  frame.frm_point <- Text.dup_point text buf.buf_point;
  frame.frm_y_offset <- 0;
  buf.buf_shared <- buf.buf_shared + 1;
  frame.frm_killed <- false
```

⟨`Efuns.frame` *other fields* 79d⟩+≡ (16a) ◁73a 81b▷
```
mutable frm_killed : bool;
```

⟨`Frame.kill()` *setting* `frm_killed` *field* 79e⟩≡ (124b)
```
frame.frm_killed <- true;
```

6.4.3 Minibuffer display

The request string ("Find file:", "M-x", etc.) is drawn at the left of the bottom line using `direct_attr` (normal video), while the minibuffer frame—which occupies the columns to its right—is painted by the regular `Frame.displayX` path like any other frame.

⟨`Frame.display()` *draw minibuffer request string* 79f⟩≡ (64c)
```
graphic.Xdraw.draw_string
 0 (top_window.top_height-1)
 request
 0 (String.length request) Text.direct_attr
```

⟨*function* `Top_window.clear_message` 79g⟩≡ (? 0—1)
```
let clear_message top_window =
  match top_window.top_mini_buffers with
  | [] ->
      let graphic = Efuns.backend top_window in
      graphic.Xdraw.clear_eol
        0 (top_window.top_height - 1)
        top_window.top_width;
  | _ -> ()
```

6.4.4 `Top_window.message()`

`Top_window.messageX` displays a transient notification (like Emacs's "Wrote foo.ml") at the bottom line. If a minibuffer is active, it forces an immediate display update and marks the minibuffer frame for redraw so the message does not permanently overwrite the prompt.

⟨*function* `Top_window.message` 80⟩≡ (? 0—1)

```
(* note: I could get rid of the top_window parameter and compute top_window
 * by getting List.hd (Globals.editors()).top_windows, but maybe
 * one day I will add back the possibility to have multiple top_windows,
 * in which case we will need to know to which top_window to display
 * a message.
 *)
let message top_window msg =
  let graphic = Efuns.backend top_window in
  let len = String.length msg in

  graphic.Xdraw.draw_string 0 (top_window.top_height - 1)
    msg 0 len Text.direct_attr;
  graphic.Xdraw.clear_eol len (top_window.top_height - 1)
    (top_window.top_width - len);

  match top_window.top_mini_buffers with
  | [] -> (* No mini-buffer ... *) ()
  | mini_buffer :: _ -> (* one mini-buffer is active *)
      graphic.Xdraw.update_display ();
      (* let _ = Unix.select [] [] [] 0.2 in *)
      mini_buffer.frm_redraw <- true
```

Chapter 7

Input Management

7.1 Keyboard

We saw in the event loop chapter how GTK events arrive and get converted to key codes. Here we see the other end of the pipeline: how a key (or key sequence) is resolved to an action function. The central function is `Top_window.try_mapX`, which accumulates prefix keys in `frm_prefix` and looks up the completed sequence through the cascading keymap chain (buffer map → minor mode maps → major mode map → global map). If the sequence is a `Prefix`, the key is stored and the user is prompted for the next key; if it is a `Function`, the action is dispatched immediately.

7.1.1 Keys to action: `try_map()`

⟨*exception* `Efuns.UnboundKey` 81a⟩≡ (? 0—1)
```
exception UnboundKey
```

⟨`Efuns.frame` *other fields* 81b⟩+≡ (16a) ◁79d 82c▷
```
mutable frm_prefix : key list;
```

⟨*function* `Top_window.try_map` 81c⟩≡ (? 0—1)
```
let try_map frame key =
  let keylist = frame.frm_prefix @ [key] in
  match Ebuffer.get_binding frame.frm_buffer keylist with
  | Function f ->
      frame.frm_prefix <- [];
      (* dispatch the action *)
      ⟨Top_window.try_map() if debug, print action name 157c⟩
      f frame;
      ⟨Top_window.try_map() set last action 82d⟩
  | Prefix _map ->
      frame.frm_prefix <- keylist;
      let top_window = Window.top frame.frm_window in
      message top_window (Keymap.print_key_list frame.frm_prefix);
  | Unbound -> raise UnboundKey
```

⟨*function* `Keymap.get_binding` 81d⟩≡ (? 0—1)
```
let rec get_binding map keylist =
  match keylist with
    [] -> Unbound
  | [key] ->
        ⟨Keymap.get_binding() find key in map 82a⟩
  | key :: tail ->
      match
        ⟨Keymap.get_binding() find key in map 82a⟩
```

```
        with
        | Prefix map -> get_binding map tail
        | _ -> Unbound
```

⟨Keymap.get_binding() *find key in map* 82a⟩≡ (81)
```
  (match key with
    (NormalMap,key) when key >= 0 && key < 256 ->
      map.char_map.(key)
  | _ ->
      try List.assoc key map.complex_bindings
      with Not_found -> Unbound
   )
```

⟨Top_window.handle_key() *handle exception of* try_map 82b⟩≡ (35d) 155c▷
```
  | UnboundKey ->
      message top_window
        (Printf.sprintf "Unbound key %s"
          (Keymap.print_key_list (frame.frm_prefix @ [key])));
      frame.frm_prefix <- [];
  (* bugfix: it is nice for Failure to have a simpler error message
   *  than for the general 'e' case below; it allows efuns plugins to
   *  give nice error messages to the user. However, in the end you want
   *  also a backtrace because non-efuns code sometimes also
   *  uses failwith (e.g., Common2.foldl1).
   * less: maybe have a Efuns.Error instead of using failwith?
   *)
   | Failure str ->
      let bt = Printexc.get_backtrace () in
      message top_window str;
      let buf = Ebuffer.default "*backtrace*" in
      let text = buf.buf_text in
      Text.insert_at_end text str;
      Text.insert_at_end text "\n";
      Text.insert_at_end text bt;
      Text.insert_at_end text "\n"

  | (Exit.ExitCode _) as x -> raise x
```

⟨Efuns.frame *other fields* 82c⟩+≡ (16a) ◁81b
```
  mutable frm_last_action : action;
```

⟨Top_window.try_map() *set last action* 82d⟩≡ (81c)
```
  frame.frm_last_action <- f
```

7.1.2 Action to keys: add_binding()

⟨*function* Keymap.add_binding 82e⟩≡ (? 0—1)
```
  let rec add_binding map key_list binding =
    match key_list with
      [] -> failwith "ERROR add_complex_binding: empty key list"
    | [key] -> set_binding map key (Function binding)
    | key :: tail ->
        match
          match key with
            (NormalMap,key) when key >= 0 && key < 256 ->
              map.char_map.(key)
          | _ ->
              try
                List.assoc key map.complex_bindings
```

```
            with
              Not_found -> Unbound
        with
          Prefix map -> add_binding map tail binding
        | Unbound ->
            let newmap = create () in
            set_binding map key (Prefix newmap);
            add_binding newmap tail binding;
        | _e ->
            failwith "ERROR add_complex_binding: Unable to add prefix"
```

⟨*function* Keymap.set_binding 83a⟩≡ (? 0—1)

```
let set_binding map key binding =
  match key with
  | (NormalMap,key) when key >= 0 && key < 256 ->
      map.char_map.(key) <- binding
  | _ ->
      map.complex_bindings <- (key,binding) :: map.complex_bindings
```

7.1.3 Basic key bindings

⟨*constant* core_map 83b⟩≡ (? 0—1)

```
let core_map =
 ⟨core_map entries 83c⟩
 []
```

⟨core_map *entries* 83c⟩≡ (83b) 83e▷

```
(* standard chars *)
(List_.enum 32 127 |> List.map (fun key ->
  [NormalMap, key], Edit.self_insert_command
)) @
```

Uses Efuns.mod_ident.ControlMap and Efuns.mod_ident.NormalMap.

⟨*function* Simple.self_insert_command 83d⟩≡ (? 0—1)

```
let self_insert_command frame =
  let char = Char.chr !Top_window.keypressed in
  let buf = frame.frm_buffer in
  ⟨Simple.self_insert_command() if overwrite mode 167a⟩
  else insert_char frame char
```

7.1.4 Entering special chars: ^M

⟨core_map *entries* 83e⟩+≡ (83b) ◁83c 152c▷

```
(let c_q = (ControlMap, Char.code 'q') in
(List_.enum 65 (65+25) |> List.map (fun key ->
  [c_q;ControlMap, key], Misc_features.insert_special_char
)) @
(* needed? *)
(List_.enum 97 (97+25) |> List.map (fun key ->
  [c_q;ControlMap, key], Misc_features.insert_special_char
))) @
```

Uses Efuns.mod_ident.ControlMap.

⟨*function* Simple.insert_special_char 83f⟩≡ (? 0—1)

```
let insert_special_char frame =
  let key = !Top_window.keypressed in
  let char = Char.chr key in
  if char >= 'a' && char <= 'z'
  then Edit.insert_char frame (Char.chr (key - 97))
  else Edit.insert_char frame (Char.chr (key - 65))
```

7.1.5 Bindings display

⟨*misc keys* 84a⟩+≡ (25) ◁72d 129d▷
```
[c_h; NormalMap, Char.code 'K'], Frame.help_bindings;
```
Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Frame.bindings_help` 84b⟩≡ (? 0—1)
```
let help_bindings (frame : t) : unit =
  let window = frame.frm_window in
  change_buffer frame.caps window "*bindings*"
[@@interactive]
```

⟨*function* `Interactive.create_bindings` 84c⟩≡ (? 0—1)
```
let create_bindings_help_buffer () =
  let s = Keymap.all_bindings () in
  let text = Text.create s in
  Ebuffer.create "*bindings*" None text (Keymap.create ())
```

⟨*function* `Keymap.all_bindings` 84d⟩≡ (? 0—1)
```
let all_bindings () =
  let s = ref "Default bindings:" in
  (* TODO *)
  !s
```

⟨*function* `Keymap.print_key` 84e⟩≡ (? 0—1)
```
let print_key (map,keysym) =
  let prefix =
    match map with
    | NormalMap -> ""
    | ControlMap -> "C-"
    | MetaMap -> "M-"
    | ControlMetaMap -> "CM-"
  in
  let kname =
    try List.assoc keysym XK.keysym_to_name
    with Not_found ->
      try Printf.sprintf "%c" (Char.chr keysym)
      with Invalid_argument _ -> "?"
  in
  prefix^kname
```

⟨*function* `Keymap.print_key_list` 84f⟩≡ (? 0—1)
```
let rec print_key_list key_list =
  match key_list with
  | [] -> ""
  | [key] -> Printf.sprintf "%s" (print_key key)
  | key :: tail ->
      Printf.sprintf "%s %s" (print_key key) (print_key_list tail)
```

7.2 Mouse

Mouse events are handled by the same key dispatch machinery as keyboard events: button presses and drags are mapped to synthetic keysyms (`xk_Pointer_Button1`, `xk_Pointer_Drag1`, etc.) and routed through `handle_keyX`. The mouse x/y coordinates are stored in globals so that the action function (e.g., `mouse_set_frameX`) can access them. This uniform treatment means mouse bindings can be customized exactly like key bindings.

⟨*constant* `Top_window.mouse_x` 84g⟩≡ (? 0—1)
```
let mouse_x = ref 0
```

⟨*constant* `Top_window.mouse_y` 85a⟩≡ (? 0—1)
```
let mouse_y = ref 0
```

⟨`Top_window.handler()` *match event cases* 85b⟩+≡ (35a) ◁35c 85c▷
```
| Xtypes.XTButtonPress (modifiers,button,x,y) ->
    mouse_x := x;
    mouse_y := y;
    handle_key top_window modifiers (XK.xk_Pointer_Button_Dflt + button)
```

⟨`Top_window.handler()` *match event cases* 85c⟩+≡ (35a) ◁85b 88c▷
```
| Xtypes.XTMouseMotion (modifiers,button,x,y) ->
    mouse_x := x;
    mouse_y := y;
    handle_key top_window modifiers (XK.xk_Pointer_Drag_Dflt + button)
```

7.2.1 Mouse navigation

Clicking in the editor must convert pixel coordinates to a buffer position—the reverse of the display pipeline. `Frame.move_pointX` takes screen (x,y), subtracts the frame's origin, and calls `Frame.cursor_to_coordX` which walks the frame table's boxes backwards: starting from the row at screen y, it iterates through the row's box list accumulating widths until it finds which box contains screen x, then divides by `box_charsize` to get the column within that box. The click also sets a mark at the old position, so subsequent mouse drags can select a region.

⟨`Config.init_global_map()` *mouse keys setup* 85d⟩≡ (24c) 87b▷
```
Keymap.add_global_key [NormalMap, XK.xk_Pointer_Button1] Mouse.mouse_set_frame;
```
Uses `Efuns.mod_ident.NormalMap` and `XK.xk_Pointer_Button3`.

⟨*function* `Simple.mouse_set_frame` 85e⟩≡ (? 0—1)
```
let mouse_set_frame frame =
  let top_window = Window.top frame.frm_window in
  let frame = Top_window.mouse_set_active top_window in
  let (buf, text, point) = Frame.buf_text_point frame in
  let mark = Ebuffer.get_mark buf point in
  Text.goto_point text mark point;
  ()
[@@interactive "set_active_frame"]
```

⟨*function* `Top_window.mouse_set_active` 85f⟩≡ (? 0—1)
```
let mouse_set_active top_window =
  let x = !mouse_x in
  let y = !mouse_y in
  let frame = find_selected_frame top_window in
  Frame.move_point frame frame.frm_point x y;
  frame
```

Frame selection

⟨*function* `Top_window.find_selected_frame` 85g⟩≡ (? 0—1)
```
let find_selected_frame top_window =
  let x = !mouse_x in
  let y = !mouse_y in
  let frame =
    if y > top_window.top_height - 2 then
      match top_window.top_mini_buffers with
      | [] -> raise Not_found
      | mini_frame :: _ -> mini_frame
```

```
      else
        find_frame top_window.window x y
    in
    Frame.active frame;
    frame
```

⟨*function* `Top_window.find_frame` 86a⟩≡ (? 0—1)
```
  let rec find_frame window x y =
    match window.win_down with
      NoFrame -> raise Not_found
    | WFrame frame -> frame
    | HComb (w1,w2) ->
        if w2.win_xpos > x
        then find_frame w1 x y
        else find_frame w2 x y
    | VComb (w1,w2) ->
        if w2.win_ypos > y
        then find_frame w1 x y
        else find_frame w2 x y
```

Point selection

⟨*function* `Frame.move_point` 86b⟩≡ (? 0—1)
```
  let move_point frame point x y =
    let (_, text, _) = buf_text_point frame in
    let coord = cursor_to_coord frame (x - frame.frm_xpos) (y - frame.frm_ypos) in
    Text.goto_line text point coord.Text.c_line;
    Text.fmove text point coord.Text.c_col
```

⟨*function* `Frame.cursor_to_point` 86c⟩≡ (? 0—1)
```
  let cursor_to_coord frame x y =
    ⟨Frame.cursor_to_point() sanity check parameters in range 86d⟩
    let frm_line = frame.frm_table.(y) in
    let line = frm_line.lineidx_in_text in
    let rec iter x boxes default =
      match boxes with
        [] -> default
      | box :: tail ->
          if x < box.box_size
          then box.box_col +   x / box.box_charsize
          else iter (x - box.box_size) tail (box.box_col + box.box_len)
    in
    let col = iter (x + frame.frm_x_offset + frm_line.first_box_extra_offset)
      frm_line.frmline_boxes 0
    in
    { Text.c_col = col; Text.c_line = line }
```

⟨`Frame.cursor_to_point()` *sanity check parameters in range* 86d⟩≡ (86c)
```
  if (y < 0) || (x<0) || (y >= frame.frm_height-1) || (x>frame.frm_cutline)
  then raise Not_found;
```

7.2.2 Mouse selection

⟨*toplevel* `Mouse` *starting hook* 86e⟩≡ (? 0—1)
```
  let _ =
    Hooks.add_start_hook (fun () ->
      Keymap.add_global_key [NormalMap, XK.xk_Pointer_Drag1] mouse_drag_region;
      Hooks.add_hook Highlight.unhighlight_hook unhighlight_hook;
    )
```

⟨*function* Simple.mouse_drag_region 87a⟩≡ (? 0—1)

```
(* C'est tout simple. On arrive dans cette fonction quand on est en train
de bouger la souris avec le bouton appuyer. La frame courante est donc
correcte. On peut utiliser la position de la souris pour trouver la
nouvelle position du curseur dans la frame. Si on en sort, on peut
ou prendre la derniere position, ou la premiere.
*)
let mouse_drag_region _frame =
  failwith "Simple.mouse_drag_region: TODO"
[@@interactive]
(*
  let top_window = Window.top frame.frm_window in
  let point = frame.frm_point in
  begin
    try
      move_point frame point !mouse_x !mouse_y
    with
      Not_found ->
        let (buf, text, _) = Frame.buf_text_point frame in
        let y = !mouse_y - frame.frm_ypos in
        if y < 0 then
          ( scroll_line frame (y-1);
            goto_point text point frame.frm_start;
            bmove text point 1; ())
        else
        if y >= frame.frm_height - 1 then
          (
            scroll_line frame (y - frame.frm_height + 2);
            goto_point text point frame.frm_end;
            fmove text point 1; ())
  end;
  highlight frame;
  let top_window = Window.top frame.frm_window in
  let xterm = top_window.top_term in
  Selection.setSelection xterm#display xterm#window XA.xa_primary
    (fun target ->
      if target = XA.xa_string then
        match !highlighted with
          None -> raise Not_found
        | Some (frame,debut,fin) ->
            let (buf, text, _) = Frame.buf_text_point frame in
            let curseur = new_point text in
            let final = new_point text in
            set_position text curseur debut;
            set_position text final fin;
            let str = Text.region text curseur final in
            remove_point text curseur;
            remove_point text final;

            1, str
      else raise Not_found
  ) !Eloop.event_time
*)
```

7.2.3 Mouse copy/paste

⟨Config.init_global_map() *mouse keys setup* 87b⟩+≡ (24c) ◁85d

```
Keymap.add_global_key [NormalMap, XK.xk_Pointer_Button3]
  Mouse.mouse_save_then_kill;
```

```
  Keymap.add_global_key [NormalMap, XK.xk_Pointer_Button2]
    Mouse.mouse_yank_at_click;
```

Uses Efuns.mod_ident.NormalMap and XK.xk_Pointer_Button2.

⟨*function* Simple.mouse_yank_at_click 88a⟩≡ (? 0—1)

```
  let  mouse_yank_at_click _frame =
  failwith "Simple.mouse_yank_at_click: TODO"
  [@@interactive "insert_at_point"]
  (*
    let top_window = Window.top frame.frm_window in
    let frame = mouse_set_active top_window in
    let (buf, text, point) = Frame.buf_text_point frame in
    let xterm = Window.xterm top_window in
    let str = WX_xterm.get_cutbuffer xterm in
    Text.insert text point str;
    Text.fmove text point (String.length str)
  *)
```

⟨*function* Simple.mouse_save_then_kill 88b⟩≡ (? 0—1)

```
  let mouse_save_then_kill _frame =
    failwith "Simple.mouse_save_then_kill: TODO"
  [@@interactive]
  (*
    let top_window = Window.top frame.frm_window in
    let frame = Top_window.find_selected_frame top_window in
    let (buf, text, point) = Frame.buf_text_point frame in
    let mark = Ebuffer.get_mark buf point in
    Text.with_new_point text (fun new_point ->
      Frame.move_point frame new_point !mouse_x !mouse_y;
      if point = new_point then begin
        let (start,term) =
          if point < mark then (point,mark) else (mark,point)
        in
        Text.delete text start (Text.distance text start term)
      end else begin
        let xterm = Window.xterm top_window in
        goto_point text mark point;
        goto_point text point new_point;
        let str = Text.region text mark point in
        kill_string str;
        WX_xterm.set_cutbuffer xterm str;
        highlight frame
      end
    )
  *)
```

7.2.4 System clipboard

7.2.5 Resize

⟨Top_window.handler() *match event cases* 88c⟩+≡ (35a) ◁85c

```
  | Xtypes.XTResize (new_width, new_height) ->
      resize_window top_window.window 0 0 new_width (new_height - 1);
      top_window.top_mini_buffers |> List.iter (fun frame ->
        let window = frame.frm_window in
        window.win_ypos <- new_height - 1;
        window.win_width <- new_width;
        Frame.install window frame
      );
```

```
    top_window.top_width <- new_width;
    top_window.top_height <- new_height;
    clear_message top_window;
    update_display ()
```

⟨*function* Top_window.resize_window 89a⟩≡ (? 0—1)
```
let rec resize_window window xpos ypos width height =
  let old_width = window.win_width in
  let old_height = window.win_height in
  window.win_xpos <- xpos;
  window.win_ypos <- ypos;
  window.win_width <- width;
  window.win_height <- height;
  match window.win_down with
    WFrame frame -> Frame.install window frame
  | NoFrame -> assert false
  | HComb (w1,w2) ->
      let wi1 = w1.win_width * width / old_width in
      resize_window w1 xpos ypos wi1 height;
      resize_window w2 (xpos + wi1) ypos (width - wi1) height
  | VComb (w1,w2) ->
      let he1 = w1.win_height * height / old_height in
      resize_window w1 xpos ypos width he1;
      resize_window w2 xpos (ypos + he1) width (height - he1)
```

7.3 Interactive select

7.3.1 M-x foo and call_interactive()

⟨*meta keys* 89b⟩≡ (25)
```
[MetaMap, Char.code 'x'], Interactive.call_interactive;
```

⟨*function* Interactive.call_interactive 89c⟩≡ (? 0—1)
```
let call_interactive frame =
  let interactives = compute_interactives () in
  Select.select frame "M-x " meta_hist ""
    (fun _ -> List.map fst interactives)
    (fun s -> s)
    (fun s -> exec_interactive interactives frame s)
[@@interactive]
```

⟨*constant* Interactive.meta_hist 89d⟩≡ (? 0—1)
```
let meta_hist = ref []
```

⟨*function* Interactive.buf_interactives 89e⟩≡ (? 0—1)
```
let compute_interactives () =
  Action.actions |> Hashtbl_.to_list |> List.map (fun (s, _) ->
    s, (Action.execute_action s, None)
  )
```

⟨*function* Interactive.exec_interactive 89f⟩≡ (? 0—1)
```
let exec_interactive interactives frame name =
  try
    let f, key = List.assoc name interactives in
    ⟨Interactive.exec_interactive() display if has a keybinding 90a⟩
    (* run it ! *)
    f frame
  with Not_found ->
    Message.message frame ("No interactive command "^name)
```

⟨Interactive.exec_interactive() *display if has a keybinding* 90a⟩≡ (89f)
```
key |> Option.iter (fun key_list ->
  Message.message frame
    ("you can run "^name^" by typing "^(Keymap.print_key_list key_list))
);
```

7.3.2 Adding interactive functions

7.3.3 `Select.select()`

`Select.selectX` is the workhorse behind all prompting operations that need completion (open file, switch buffer, execute command, etc.). It creates a minibuffer with a custom keymap that intercepts Tab for completion and up/down for history navigation. On Tab, it computes the common suffix of all completions; if there are multiple matches, it displays them in a "`*completions*`" buffer. The `incremental_mini_buffer` variant additionally calls an `incremental_action` callback on every keystroke, which is how incremental search (`C-s`) provides live feedback.

⟨*function* `Select.select` 90b⟩≡ (? 0—1)
```
let select frame request history start completion_fun prefix_fun action =
  let map = Keymap.create () in
  let string = ref "" in
  Keymap.add_binding map [ControlMap, Char.code 'g'] (fun mini_frame ->
    remove_completions mini_frame;
    Minibuffer.kill mini_frame frame
  );
  let completion = ref "_*NoCompletion*_" in
  let completions = ref [] in
  Keymap.add_binding map [NormalMap, XK.xk_Tab] (fun mini_frame ->
      (* TODO? was != but I think it should be <> *)
      if (Eq.phys_not_equal !completion !string) then begin
        let text = mini_frame.frm_buffer.buf_text in
        completions := completion_fun !string;
        let suffix, n  = Utils.common_suffix !completions (prefix_fun !string)in
        completion := !string;
        Text.insert text mini_frame.frm_point suffix;
        Text.fmove text mini_frame.frm_point (String.length suffix);
        if n>1
        then display_completions frame
                (Utils.completion !completions (prefix_fun !string));

      end else
      (* can be improved, by displaying another part of the completion buffer *)
        display_completions frame
          (Utils.completion !completions (prefix_fun !string))
  );
  set_history map string history;

  incremental_mini_buffer frame map request start
    (fun _frame str -> string := str;)
    (fun _old_frame str ->
       (match !history with
        | hd :: _ when hd = str -> ()
        | _ ->
            if str <> ""
            then history := str :: !history
        );
       remove_completions frame;
       action str
```

```
  ) |> ignore
```

7.3.4 Incremental mini buffer

⟨*function* `Select.incremental_mini_buffer` 91a⟩≡ (? 0—1)

```
let incremental_mini_buffer
 frame ismap request default incremental_action end_action =
  let top_window = Window.top frame.frm_window in

  let incremental mini_frame =
    incremental_action frame (Text.to_string mini_frame.frm_buffer.buf_text)
  in
  let incremental_insert mini_frame =
    Edit.self_insert_command mini_frame;
    incremental mini_frame
  in

  for key = 32 to 127 do
    Keymap.add_binding ismap [NormalMap, key] incremental_insert
  done;
  Keymap.add_binding ismap [NormalMap, XK.xk_BackSpace] (fun mini_frame ->
    Edit.delete_backspace_char mini_frame;
    incremental mini_frame
  );

  top_window.top_second_cursor <- Some frame;
  Minibuffer.create_return frame ismap request default (fun frame str ->
    top_window.top_second_cursor <- None;
    end_action frame str
  )
```

⟨*function* `Minibuffer.create_return` 91b⟩≡ (? 0—1)

```
let create_return frame local_map request default action =
  let mini_frame = create frame local_map request in
  Edit.insert_string mini_frame default;
  Keymap.add_binding local_map [NormalMap, XK.xk_Return]
    (return action frame);
  mini_frame
```

⟨*function* `Minibuffer.return` 91c⟩≡ (? 0—1)

```
let return action old_frame = fun mini_frame ->
  let repstr = Text.to_string mini_frame.frm_buffer.buf_text in
  kill mini_frame old_frame;
  action old_frame repstr
```

⟨*function* `Text.to_string` 91d⟩≡ (? 0—1)

```
let to_string text =
  let len = text.text_size - text.gsize in
  if len = 0
  then ""
  else begin
    let str = Bytes.create len in
    let gpos = text.gpoint.pos in
    let gap_end = gpos + text.gsize in
    Bytes.blit text.text_string 0 str 0 gpos;
    Bytes.blit text.text_string gap_end str gpos (len- gpos);
    Bytes.to_string str
  end
```

7.3.5 Secondary cursor

⟨Efuns.top_window *other fields* 92a⟩+≡ (19b) ◁77a
```
mutable top_second_cursor : frame option;
```

⟨Top_window.cursor_on() *display secondary cursor* 92b⟩≡ (69a)
```
top_window.top_second_cursor |> Option.iter (fun frame ->
  set_cursor_on top_window frame
)
```

⟨Top_window.cursor_off() *hide secondary cursor* 92c⟩≡ (71b)
```
top_window.top_second_cursor |> Option.iter (fun frame ->
  set_cursor_off top_window frame
)
```

⟨clean_display() *sanity check second cursor and minibuffer* 92d⟩≡ (71a)
```
(match top_window.top_mini_buffers, top_window.top_second_cursor with
 | _, None -> ()
 | _::_, Some _ -> ()
 | [], Some frame ->
     Error.error
      (spf "clean_display: weird, second cursor for %s but no minibuffer"
       (frame.frm_buffer.buf_name));
     top_window.top_second_cursor <- None
)
```

7.3.6 Interactive variables

⟨*constant* Complex.value_hist 92e⟩≡ (? 0—1)
```
let value_hist = ref []
```

⟨*constant* Complex.all_vars 92f⟩≡ (? 0—1)
```
let all_vars = ref None
```

⟨*function* Complex.all_variables 92g⟩≡ (? 0—1)
```
let all_variables frame _ =
  let buf = frame.frm_buffer in
  match !all_vars with
    Some (f,l) when Eq.phys_equal f frame -> l
  | _ ->
      let list =
        (Store.list buf.buf_vars) @
        (Store.list (Globals.editor()).edt_vars)
      in
      all_vars := Some (frame, list);
      list
```

⟨*function* Complex.set_local_variable 92h⟩≡ (? 0—1)
```
let set_local_variable frame =
  Select.select frame "set_local_variable : " variable_hist
    "" (all_variables frame) (fun s -> s) (fun variable ->
      Select.select_string frame (Printf.sprintf "%s : " variable)
      value_hist "" (fun value ->
          Store.input frame.frm_buffer.buf_vars variable value))
[@@interactive]
```

⟨*function* `Complex.set_global_variable` 93⟩≡ (? 0—1)

```
let set_global_variable frame =
  Select.select frame "set_global_variable : " variable_hist
    "" (all_variables frame) (fun s -> s) (fun variable ->
      Select.select_string frame (Printf.sprintf "%s : " variable)
      value_hist "" (fun value ->
          Store.input (Globals.editor()).edt_vars variable value))
[@@interactive]
```

Chapter 8

Basic Features

8.1 Navigating

⟨*navigating keys* 94a⟩≡ (25) 100f▷
```
[NormalMap, XK.xk_Left], (fun frm -> ignore (Move.move_backward frm 1));
[NormalMap, XK.xk_Right], (fun frm -> ignore (Move.move_forward frm 1));
[ControlMap, Char.code 'b'], (fun frm -> ignore (Move.move_backward frm 1));
[ControlMap, Char.code 'f'], (fun frm -> ignore (Move.move_forward frm 1));

[ControlMap, XK.xk_Left ], (Frame.to_frame Move.backward_word);
[ControlMap, XK.xk_Right ], (Frame.to_frame Move.forward_word);
(* ?? standard ?*)
[MetaMap, XK.xk_Left ], (Frame.to_frame Move.backward_word);
[MetaMap, XK.xk_Right ], (Frame.to_frame Move.forward_word);

[ControlMap, Char.code 'a'], Move.beginning_of_line;
[ControlMap, Char.code 'e'], Move.end_of_line;

[NormalMap, XK.xk_Up], Move.backward_line;
[NormalMap, XK.xk_Down], Move.forward_line;


[ControlMap, XK.xk_Up], (Frame.to_frame Move.backward_paragraph);
[ControlMap, XK.xk_Down], (Frame.to_frame Move.forward_paragraph);

[NormalMap, XK.xk_Prior], Scroll.backward_screen;
[NormalMap, XK.xk_Next], Scroll.forward_screen;

[ControlMap, XK.xk_Next], Move.end_of_file;
[ControlMap, XK.xk_Prior], Move.begin_of_file;
[MetaMap, Char.code '>'], Move.end_of_file;
[MetaMap, Char.code '<'], Move.begin_of_file;
```
Uses Efuns.mod_ident.ControlMap, Efuns.mod_ident.MetaMap, Efuns.mod_ident.NormalMap, XK.xk_Down, XK.xk_Left, XK.xk_Next, XK.xk_Prior, XK.xk_Right, and XK.xk_Up.

8.1.1 Characters

Character movement is the simplest case: move the point forward or backward by a delta, returning how far it actually moved (which may be less at buffer boundaries thanks to `fmove_res`/`bmove_res`).

⟨*function* `Simple.move_backward` 94b⟩≡ (? 0—1)
```
let move_backward frame delta =
  let (_buf, text, point) = Frame.buf_text_point frame in
  Text.bmove text point delta
```

⟨*function* `Simple.move_forward` 95a⟩≡ (? 0—1)
```
let move_forward frame delta =
  let (_buf, text, point) = Frame.buf_text_point frame in
  Text.fmove text point delta
```

8.1.2 Words and the syntax table

A syntax table is a 256-entry boolean array indicating which byte values are "word constituents." By default only letters and digits qualify, but major modes customize this—for instance, OCaml mode adds `_` and `'` so that identifiers like `my_var'` are treated as single words. The word movement functions (`backward_word`, `forward_word`) use two helper loops: `in_prev_word` skips non-word characters, then `to_begin_of_word` walks backward through word characters. Emacs's syntax table is richer (it distinguishes punctuation, string delimiters, comment starters, etc.), but this simpler boolean version suffices for word-level operations.

⟨`Efuns.buffer` *other fields* 95b⟩+≡ (15a) ◁53b 103b▷
```
mutable buf_syntax_table : bool array;
```
Uses `Efuns.map`.

⟨*function* `Ebuffer.create_syntax_table` 95c⟩≡ (? 0—1)
```
let create_syntax_table ()  =
  let table = Array.make 256 false
  in
  for i = Char.code 'a' to Char.code 'z' do
    table.(i) <- true;
  done;
  for i = Char.code 'A' to Char.code 'Z' do
    table.(i) <- true;
  done;
  for i = Char.code '0' to Char.code '9' do
    table.(i) <- true;
  done;
  table
```

⟨*constant* `Ebuffer.default_syntax_table` 95d⟩≡ (? 0—1)
```
let default_syntax_table = create_syntax_table ()
```

⟨*function* `Simple.backward_word` 95e⟩≡ (? 0—1)
```
let backward_word buf point =
  let text = buf.buf_text in
  let syntax = buf.buf_syntax_table in
  in_prev_word text point syntax;
  to_begin_of_word text point syntax
```

⟨*function* `Simple.in_prev_word` 95f⟩≡ (? 0—1)
```
let in_prev_word text mark syntax =
  while Text.bmove_res text mark 1 = 1 &&
        (not syntax.(Char.code (Text.get_char text mark)))
  do () done
```

⟨*function* `Simple.to_begin_of_word` 95g⟩≡ (? 0—1)
```
let to_begin_of_word text mark syntax =
  if Text.bmove_res text mark 1 = 1 then
    begin
      while syntax.(Char.code (Text.get_char text mark)) &&
            (Text.bmove_res text mark 1) <> 0
      do () done;
      if not syntax.(Char.code (Text.get_char text mark))
      then (Text.fmove text mark 1)
    end
```

⟨*function* `Simple.forward_word` 96a⟩≡ (? 0—1)

```
let forward_word buf point =
  let text = buf.buf_text in
  let syntax = buf.buf_syntax_table in
  in_next_word text point syntax;
  to_end_of_word text point syntax
```

⟨*function* `Simple.in_next_word` 96b⟩≡ (? 0—1)

```
let in_next_word text mark syntax =
  while (not syntax.(Char.code (Text.get_char text mark))) &&
        Text.fmove_res text mark 1 = 1
  do () done
```

⟨*function* `Simple.to_end_of_word` 96c⟩≡ (? 0—1)

```
let to_end_of_word text mark syntax =
  while syntax.(Char.code (Text.get_char text mark)) &&
       (Text.fmove_res text mark 1) <> 0
  do () done
```

⟨*function* `Simple.end_of_word` 96d⟩≡ (? 0—1)

```
let end_of_word  buf point =
  let text = buf.buf_text in
  let syntax = buf.buf_syntax_table in
  Text.with_dup_point text point (fun mark ->
    to_end_of_word text mark syntax;
    let s = Text.region text point mark in
    s
  )
```

⟨*function* `Simple.beginning_of_word` 96e⟩≡ (? 0—1)

```
let beginning_of_word buf point =
  let text = buf.buf_text in
  let syntax = buf.buf_syntax_table in
  Text.with_dup_point text point (fun mark ->
    to_begin_of_word text mark syntax;
    let s = Text.region text mark point in
    s
  )
```

8.1.3 Lines

Vertical movement is trickier than it looks because of the goal column problem. When you press Down repeatedly, Emacs keeps the cursor at the column where the run of vertical moves started, even if intermediate lines are shorter. The author's comments record the struggle to implement this: a naive approach (save column, move, restore) breaks when other commands intervene. The solution, borrowed from Microemacs, uses `frm_last_action`: if the previous action was `forward_line` or `backward_line`, reuse the saved `temporary_goal_column`; otherwise, compute a fresh one. The `forward_line` and `backward_line` functions are mutually recursive with `goal_column` because they all reference each other.

⟨*function* `Simple.beginning_of_line` 96f⟩≡ (? 0—1)

```
let beginning_of_line frame =
  move_backward frame (begin_to_point frame) |> ignore
[@@interactive]
```

⟨*function* `Simple.begin_to_point` 96g⟩≡ (? 0—1)

```
let begin_to_point frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.point_to_bol text point
```

⟨*function* Simple.end_of_line 97a⟩≡ (? 0—1)
```
let end_of_line frame =
  move_forward frame (point_to_end frame) |> ignore
[@@interactive]
```

⟨*function* Simple.point_to_end 97b⟩≡ (? 0—1)
```
let point_to_end frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.point_to_eol text point
```

⟨*function* Simple.backward_line 97c⟩≡ (? 0—1)
```
and backward_line frame =
  let text = frame.frm_buffer.buf_text in
  let point = frame.frm_point in
  if Text.point_line text point > 0 then begin
    let goal_col = goal_column frame in
    beginning_of_line frame;
    move_backward frame 1 |> ignore;
    move_to_goal_column frame goal_col;
  end
```

⟨*function* Simple.forward_line 97d⟩≡ (? 0—1)
```
and forward_line frame =
  let text = frame.frm_buffer.buf_text in
  let point = frame.frm_point in
  if Text.point_line text point < Text.nbr_lines text then begin
    let goal_col = goal_column frame in
    end_of_line frame;
    move_forward frame 1 |> ignore;
    move_to_goal_column frame goal_col;
  end
```

⟨*constant* Simple.temporary_goal_column 97e⟩≡ (? 0—1)
```
let temporary_goal_column =
  Store.create_abstr "Simple.temporary_goal_column"
```

⟨*function* Simple.goal_column 97f⟩≡ (? 0—1)
```
let rec goal_column frame =
  let cur_col = begin_to_point frame in
  if frame.frm_last_action == forward_line ||
     frame.frm_last_action == backward_line
  then
    try Var.get_local frame.frm_buffer temporary_goal_column
    with Not_found -> cur_col
  else cur_col
```

⟨*function* Simple.move_to_goal_column 97g⟩≡ (? 0—1)
```
and move_to_goal_column frame goal_col =
  move_backward frame (begin_to_point frame) |> ignore;
  move_forward frame (min goal_col (point_to_end frame)) |> ignore;
  Var.set_local frame.frm_buffer temporary_goal_column goal_col
```

8.1.4 Paragraphs

⟨*function* Simple.backward_paragraph 97h⟩≡ (? 0—1)
```
let backward_paragraph buf point =
  let text = buf.buf_text in
  while Text.bmove_res text point 1 = 1 &&
        (let c = Text.get_char text point in c = '\n' || c = ' ')
```

```
    do () done;
    try
      Text.search_backward text (Str.regexp "\n *\n") point |> ignore;
      Text.fmove text point 1
    with Not_found -> Text.set_position text point 0
```

⟨*function* `Simple.forward_paragraph` 98a⟩≡ (? 0—1)

```
  let forward_paragraph buf point =
    let text = buf.buf_text in
    while (let c = Text.get_char text point in c = '\n' || c = ' ') &&
          Text.fmove_res text point 1 = 1
    do () done;
    try
      Text.search_forward text (Str.regexp "\n *\n") point |> ignore;
      Text.fmove text point 1
    with Not_found -> Text.set_position text point (Text.size text)
```

8.1.5 Screens

Page-up and page-down do not move the buffer point at all—they adjust `frm_y_offset`, a virtual scroll displacement that `Frame.displayX` uses to shift which text line corresponds to the top of the frame. Setting `frm_force_start` tells the display engine to honor the new offset rather than auto-centering on the cursor. The "−2" overlap keeps two lines of context visible across pages, matching standard Emacs behavior.

⟨*function* `Simple.backward_screen` 98b⟩≡ (? 0—1)

```
  let backward_screen frame =
    frame.frm_force_start <- true;
    frame.frm_redraw <- true;
    frame.frm_y_offset <- frame.frm_y_offset - frame.frm_height + 2
```

⟨*function* `Simple.forward_screen` 98c⟩≡ (? 0—1)

```
  let forward_screen frame =
    frame.frm_force_start <- true;
    frame.frm_redraw <- true;
    frame.frm_y_offset <- frame.frm_y_offset + frame.frm_height - 2
```

⟨*function* `Simple.scroll_line` 98d⟩≡ (? 0—1)

```
  let scroll_line frame n =
    frame.frm_force_start <- true;
    frame.frm_redraw <- true;
    frame.frm_y_offset <- frame.frm_y_offset + n
```

8.1.6 File

⟨*function* `Simple.begin_of_file` 98e⟩≡ (? 0—1)

```
  let begin_of_file frame =
    let (_, text, point) = Frame.buf_text_point frame in
    ⟨save current pos from frame for position history navigation (in simple.ml) ??⟩
    Text.set_position text point 0
  [@@interactive]
```

⟨*function* `Simple.end_of_file` 98f⟩≡ (? 0—1)

```
  let end_of_file frame =
    let (_, text, point) = Frame.buf_text_point frame in
    ⟨save current pos from frame for position history navigation (in simple.ml) ??⟩
    Text.set_position text point (Text.size text)
  [@@interactive]
```

8.1.7 Specific positions

⟨*function* Complex.goto_char 99a⟩≡ (? 0—1)
```
let goto_char frame =
  Select.simple_select frame "goto-char:" (fun name ->
    let char = int_of_string name in
    Text.set_position frame.frm_buffer.buf_text frame.frm_point char
  )
[@@interactive]
```

⟨*function* Complex.goto_line 99b⟩≡ (? 0—1)
```
let goto_line frame =
  Select.simple_select frame "goto-line:" (fun name ->
    let line = int_of_string name in
    ⟨save current pos from frame for position history navigation 100d⟩
    Text.goto_line frame.frm_buffer.buf_text frame.frm_point (line - 1)
  )
[@@interactive]
```

⟨*function* Text.goto_line 99c⟩≡ (? 0—1)
```
let goto_line text point y =
  if text.text_nlines - 1 <= y
  then set_position text point (size text)
  else begin
    let line = text.text_newlines.(y) in
    point.pos <- line.position;
    point.line <- y
  end
```

⟨*function* Text.point_to_eol 99d⟩≡ (? 0—1)
```
let point_to_eol text point =
  low_distance text point.pos
    (text.text_newlines.(point.line + 1).position - 1)
```

⟨*function* Text.point_to_bol 99e⟩≡ (? 0—1)
```
(* less: kinda bol_to_point really *)
let point_to_bol text point =
  low_distance text
    text.text_newlines.(point.line).position
    point.pos
```

⟨*function* Text.point_to_eof 99f⟩≡ (? 0—1)
```
let point_to_eof text point =
  low_distance text point.pos text.text_size
```

⟨*function* Text.point_to_bof 99g⟩≡ (? 0—1)
```
let point_to_bof text point =
  low_distance text 0 point.pos
```

⟨*function* Text.point_to_lof 99h⟩≡ (? 0—1)
```
let point_to_lof text point n =
  if n > 0
  then point_to_eof text point
  else point_to_bof text point
```

⟨*function* Text.point_to_lol 99i⟩≡ (? 0—1)
```
let point_to_lol text point n =
  if n > 0
  then point_to_eol text point
  else point_to_bol text point
```

⟨*function* `Text.point_to_line` 100a⟩≡ (? 0—1)
```
let point_to_line text point line =
  let pos =
    if text.text_nlines <= line + 1
    then text.text_size
    else text.text_newlines.(line).position
  in
  move_point_to_pos text point pos
```

8.1.8 Positions history

Undoing a page scroll is easy (just scroll back), but undoing a jump like `M->` or `C-s` is not—the user has no idea where they came from. The position history solves this: commands that perform large jumps (goto-line, search, begin/end of file) call `save_current_pos` first, which pushes a duplicate point onto a fixed-size ring buffer (`buf_history_pos`). `C-u C-SPC` pops the most recent entry and moves back. The ring is capped at 10 entries, and old points must be explicitly `remove_point`ed to avoid leaking registered points in the text layer.

⟨`Efuns.buffer` *history fields* 100b⟩+≡ (15a) ◁59d
```
mutable buf_history_pos: Text.point option array;
```

⟨*constant* `Simple.history_pos_max` 100c⟩≡ (? 0—1)
```
let history_pos_max = 10
```

⟨*save current pos from frame for position history navigation* 100d⟩≡ (115f 110f 103c 99b)
```
Move.save_current_pos frame;
```

⟨*function* `Simple.save_current_pos` 100e⟩≡ (? 0—1)
```
let save_current_pos frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  if Array.length buf.buf_history_pos < history_pos_max
  then buf.buf_history_pos <- Array.make history_pos_max None;
  let arr = buf.buf_history_pos in
  arr.(history_pos_max -1) |> Option.iter (fun pt ->
     Text.remove_point text pt
  );
  Array.blit arr 0 arr 1 (history_pos_max - 1);
  arr.(0) <- Some (Text.dup_point buf.buf_text point)
```

⟨*navigating keys* 100f⟩+≡ (25) ◁94a 106a▷
```
[ControlMap, Char.code 'u'; ControlMap, Char.code ' '],
  Move.goto_last_saved_pos;
```
Uses `Efuns.mod_ident.ControlMap`.

⟨*function* `Simple.goto_last_saved_pos` 100g⟩≡ (? 0—1)
```
let goto_last_saved_pos frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  if Array.length buf.buf_history_pos < history_pos_max
  then buf.buf_history_pos <- Array.make history_pos_max None;
  let arr = buf.buf_history_pos in
  let head = arr.(0) in
  Array.blit arr 1 arr 0 (history_pos_max - 1);
  arr.(history_pos_max - 1) <- None;
  match head with
  | Some pt -> Text.goto_point text point pt
  | None -> failwith "No position history"
[@@interactive]
```

8.2 Inserting

⟨*inserting keys* 101a⟩≡ (25)
```
[NormalMap, XK.xk_Return], Edit.insert_return;
```

⟨*function* Simple.insert_return 101b⟩≡ (? 0—1)
```
let insert_return frame =
  insert_char frame '\n'
[@@interactive]
```

⟨*constant* Simple.single_char 101c⟩≡ (? 0—1)
```
let single_char = Bytes.make 1 ' '
```

⟨*function* Simple.insert_char 101d⟩≡ (? 0—1)
```
(* could factorize and just call insert_string? *)
let insert_char frame char =
  let (_, text, point) = Frame.buf_text_point frame in
  Bytes.set single_char 0 char;
  Text.insert text point (Bytes.to_string single_char);
  Text.fmove text point 1
```

⟨*function* Text.insert 101e⟩≡ (? 0—1)
```
let insert text point str =
   insert_res text point str |> ignore
```

⟨*function* Text.insert_res 101f⟩≡ (? 0—1)
```
let insert_res text point str =
  let (pos,len,modif) = low_insert text point.pos str in
  text.text_history <- Insertion(pos,len,modif) :: text.text_history;
  text.text_modified <- text.text_modified + 1;
  pos, len
```

8.3 Deleting

⟨*deleting keys* 101g⟩≡ (25) 150a▷
```
[NormalMap, XK.xk_BackSpace], Edit.delete_backspace_char;
[ControlMap, Char.code 'd'], Edit.delete_char;
[NormalMap, XK.xk_Delete], Edit.delete_char;

[MetaMap, Char.code 'd' ], (Frame.to_frame Edit.delete_forward_word);
[MetaMap, XK.xk_BackSpace ], (Frame.to_frame Edit.delete_backward_word);

[ControlMap, Char.code 'k'], Copy_paste.kill_end_of_line;
```
Uses Efuns.mod_ident.ControlMap, Efuns.mod_ident.MetaMap, Efuns.mod_ident.NormalMap, XK.xk_BackSpace, and XK.xk_Delete.

8.3.1 Characters

⟨*function* Simple.delete_backspace_char 101h⟩≡ (? 0—1)
```
let delete_backspace_char frame =
  let (_, text, point) = Frame.buf_text_point frame in
  if Text.bmove_res text point 1 <> 0
  then Text.delete text point 1
[@@interactive]
```

⟨*function* `Simple.delete_char` 102a⟩≡ (? 0—1)
```
let delete_char frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.delete text point 1
[@@interactive]
```

⟨*function* `Text.delete` 102b⟩≡ (? 0—1)
```
let delete text point len =
  delete_res text point len |> ignore
```

⟨*function* `Text.delete_res` 102c⟩≡ (? 0—1)
```
let delete_res text point len =
  let (pos,str,modif) = low_delete text point.pos len in
  text.text_history <- Deletion(pos,str,modif) :: text.text_history;
  text.text_modified <- text.text_modified + 1;
  pos, str
```

8.3.2 Words

⟨*function* `Simple.delete_backward_word` 102d⟩≡ (? 0—1)
```
let delete_backward_word buf point =
  let text = buf.buf_text in
  Text.with_dup_point text point (fun old_point ->
    backward_word buf point;
    Text.delete text point (Text.distance text point old_point)
  )
```

⟨*function* `Simple.delete_forward_word` 102e⟩≡ (? 0—1)
```
let delete_forward_word buf point =
  let text = buf.buf_text in
  Text.with_dup_point text point (fun old_point ->
    forward_word buf point;
    let len = Text.distance text old_point point in
    Text.bmove text point len;
    Text.delete text point len
  )
```

8.3.3 Lines

⟨*function* `Simple.kill_end_of_line` 102f⟩≡ (? 0—1)
```
let kill_end_of_line frame =
  let (_, text, _) = Frame.buf_text_point frame in
  let eol = point_to_end frame in
  let len =
    (* if already at eol then kill the newline *)
    if eol = 0
    then 1
    else eol
  in
  kill_text text frame.frm_point len
[@@interactive]
```

8.4 Copying and pasting

Copy and paste revolves around the kill ring, a circular array of recently killed (cut) strings. `C-w` (`kill_region`) deletes the text between point and mark and pushes it onto the ring; `M-w` (`copy_region`) copies without deleting.

`C-y` (`insert_killed`) yanks the most recent entry, and `M-y` (`insert_next_killed`) cycles through older entries by replacing the just-yanked text. This is possible because `last_insert` records the frame, position, and length of the last yank, so `M-y` can delete it before inserting the next ring entry.

⟨*moving keys* 103a⟩≡ (25)
```
[ControlMap, Char.code ' '], Copy_paste.mark_at_point;
[ControlMap, Char.code 'w'], Copy_paste.kill_region;
[ControlMap, Char.code 'y'], Copy_paste.insert_killed;
[MetaMap, Char.code 'y'], Copy_paste.insert_next_killed;
[MetaMap, Char.code 'w'], Copy_paste.copy_region;
[c_x; NormalMap, Char.code 'h'], Copy_paste.mark_whole_buffer;
```
Uses `Efuns.mod_ident.ControlMap`, `Efuns.mod_ident.MetaMap`, and `Efuns.mod_ident.NormalMap`.

8.4.1 Mark

⟨`Efuns.buffer` *other fields* 103b⟩+≡ (15a) ◁95b 126a▷
```
mutable buf_mark : Text.point option;
```
Uses `Efuns.buffer`.

⟨*function* `Complex.mark_at_point` 103c⟩≡ (? 0—1)
```
let mark_at_point frame =
  Ebuffer.set_mark frame.frm_buffer frame.frm_point;
  ⟨save current pos from frame for position history navigation 100d⟩
  Message.message frame "Mark set";
  ()
[@@interactive]
```

⟨*function* `Ebuffer.set_mark` 103d⟩≡ (? 0—1)
```
let set_mark buf point =
  let text = buf.buf_text in
  buf.buf_modified <- buf.buf_modified + 1;
  match buf.buf_mark with
  | None ->
      let mark = Text.dup_point text point in
      buf.buf_mark <- Some mark
  | Some mark ->
      Text.goto_point text mark point
```

⟨*function* `Ebuffer.get_mark` 103e⟩≡ (? 0—1)
```
let rec get_mark buf point =
  match buf.buf_mark with
  | None ->
      set_mark buf point;
      get_mark buf point
  | Some mark -> mark
```

⟨*function* `Ebuffer.remove_mark` 103f⟩≡ (? 0—1)
```
let remove_mark buf =
  buf.buf_mark |> Option.iter (fun mark ->
    buf.buf_mark <- None;
    Text.remove_point buf.buf_text mark;
    buf.buf_modified <- buf.buf_modified + 1
  )
```

8.4.2 Cut

⟨*function* Simple.kill_region 104a⟩≡ (? 0—1)
```
let kill_region frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  let mark =
    match buf.buf_mark with
    | None -> failwith "No mark set"
    | Some mark -> mark
  in
  let (start,term) =
    if mark > point then (point,mark) else (mark,point)
  in
  let _,region = Text.delete_res text start (Text.distance text start term) in
  (* less: would be better to do that in kill_string *)
  add_clipboard frame region;
  kill_string region
[@@interactive]
```

⟨*constant* Simple.kill_ring 104b⟩≡ (? 0—1)
```
let kill_ring = Array.make kill_max ""
```

⟨*constant* Simple.kill_max 104c⟩≡ (? 0—1)
```
let kill_max = 10
```

⟨*constant* Simple.kill_size 104d⟩≡ (? 0—1)
```
let kill_size = ref 0
```

⟨*function* Simple.kill_string 104e⟩≡ (? 0—1)
```
let kill_string str =
  Array.blit kill_ring 0 kill_ring 1 (kill_max - 1);
  incr kill_size;
  kill_ring.(0) <- str
```

8.4.3 Copy

⟨*function* Simple.copy_region 104f⟩≡ (? 0—1)
```
(* copy-region-as-kill-nomark in emacs *)
let copy_region frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  let mark =
    match buf.buf_mark with
    | None -> failwith "No mark set"
    | Some mark ->
        buf.buf_mark <- None;
        mark
  in
  let (start,term) =
    if mark > point then (point,mark) else (mark,point)
  in
  let region = Text.sub text start (Text.distance text start term) in
  add_clipboard frame region;
  kill_string region;
  (* need a message because as opposed to a cut this is not directly visible *)
  Message.message frame "Region saved"
[@@interactive]
```

⟨*function* Text.sub 104g⟩≡ (? 0—1)
```
let sub text point len =
  let str = Bytes.create len in
  blit str text point len |> ignore;
  Bytes.to_string str
```

⟨*function* Text.blit 105a⟩≡ (? 0—1)
```
let blit str text point len =
  let pos = point.pos in
  let len = min len (low_distance text pos text.text_size) in
  let gpos = text.gpoint.pos in
  let gap_end = gpos + text.gsize in
  if pos+len >= gpos && pos < gap_end
  then clean_text text;
  (try
     Bytes.blit text.text_string pos str 0 len
   with e -> raise e
   );
  len
```

⟨*function* Text.region 105b⟩≡ (? 0—1)
```
let rec region text p1 p2 =
  if p1>p2
  then region text p2 p1
  else sub text p1 (distance text p1 p2)
```

8.4.4 Paste

⟨*function* Simple.insert_killed 105c⟩≡ (? 0—1)
```
let insert_killed frame =
  let (_, text, point) = Frame.buf_text_point frame in
(*
  let top_window = Window.top frame.frm_window in
  let graphic = Efuns.backend top_window in
  let str =
    match graphic.Xdraw.get_clipboard () with
    | None | Some "" -> kill_ring.(0)
    | Some s ->
      graphic.Xdraw.set_clipboard None;
      s
  in
*)
  let str = kill_ring.(0) in

  let pos, len =  Text.insert_res text point str in
  Text.fmove text point len;
  last_insert := Some(frame,pos,0,len)
[@@interactive]
```

⟨*constant* Simple.last_insert 105d⟩≡ (? 0—1)
```
let last_insert = ref None
```

⟨*function* Simple.insert_next_killed 105e⟩≡ (? 0—1)
```
let insert_next_killed frame =
  let (_, text, point) = Frame.buf_text_point frame in
  match !last_insert with
  |  Some (oldframe,oldpoint,n,len) when
        oldframe == frame && oldpoint + len = Text.get_position text point ->
      let n =
        if n = (min !kill_size kill_max) - 1
        then 0
        else n+1
      in
      Text.bmove text point len;
      Text.delete text point len;
```

```
      let pos, len =  Text.insert_res text point kill_ring.(n) in
      Text.fmove text point len;
      last_insert := Some(frame,pos,n,len)
  | _ -> ()
[@@interactive]
```

8.4.5 Mark navigation

⟨*navigating keys* 106a⟩+≡ (25) ◁100f
```
[c_x; ControlMap, Char.code 'x'], Move.point_at_mark;
```

⟨*function* `Simple.point_at_mark` 106b⟩≡ (? 0—1)
```
let point_at_mark frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  let mark = Ebuffer.get_mark buf point in
  let pos = Text.get_position text point in
  Text.goto_point text point mark;
  Text.set_position text mark pos
[@@interactive]
```

8.4.6 Highlighting the selection

8.4.7 Clipboard

8.5 Undo

Undo in efuns is built into the text layer: every `low_insert` and `low_delete` appends an `Insertion` or `Deletion` action to the `text_history` list, recording the position, content, and modification counter. `Text.undoX` pops the most recent action and applies the inverse operation (delete undoes an insertion, insert undoes a deletion). The `Session` variant groups multiple actions so that a compound operation (e.g., a search-and-replace step) can be undone as a single unit.

⟨*undoing keys* 106c⟩≡ (25)
```
[ControlMap, Char.code '_'], Edit.undo;
```

⟨*function* `Simple.undo` 106d⟩≡ (? 0—1)
```
let undo frame =
  let (_, text, point) = Frame.buf_text_point frame in
  let _action, at_point, len = Text.undo text in
  frame.frm_last_text_updated <- Text.version text - 1;
  ⟨save current pos from frame for position history navigation (in simple.ml) ??⟩
  Text.set_position text point at_point;
  Text.fmove text point len
[@@interactive]
```

⟨*type* `Text.action` 106e⟩≡ (? 0—1)
```
and action =
  Insertion of position * int * int
| Deletion of position * string * int
⟨Text.action other cases 107c⟩
```

⟨*function* Text.undo 107a⟩≡ (? 0—1)
```
let undo text =
  let rec undo action =
    let gpos = text.gpoint.pos in
    let gsize = text.gsize in
    match action with
    | Insertion(point_pos, len, modified) ->
        let point = if point_pos > gpos then point_pos + gsize else point_pos in
        let (pos,str,modif) = low_delete text point len in
        text.text_modified <- modified;
        Deletion(pos,str,modif), point_pos, 0
    | Deletion (point_pos, str, modified) ->
        let point = if point_pos > gpos then point_pos + gsize else point_pos in
        let (pos,len,modif) = low_insert text point str in
        text.text_modified <- modified;
        Insertion(pos,len,modif), point_pos, String.length str
    ⟨Text.undo() match action cases 107e⟩
  in
  match text.text_history with
    [] -> raise Not_found
  | action :: tail ->
      let rev_action = undo action in
      text.text_history <- tail;
      rev_action
```

⟨Text.t *history fields* 107b⟩+≡ (13a) ◁14a
```
mutable text_history : action list;
```

⟨Text.action *other cases* 107c⟩≡ (106e)
```
| Session of action list
```

⟨*type* Text.session 107d⟩≡ (? 0—1)
```
type session = int
```

⟨Text.undo() *match action cases* 107e⟩≡ (107a)
```
| Session actions ->
    let last_point = ref 0 in
    let last_len = ref 0 in
    let rev_actions =
      List.fold_left (fun undos action ->
          let (rev_action,point,len) = undo action in
          last_point := point;
          last_len := len;
          rev_action :: undos
      ) [] actions
    in
    Session rev_actions, !last_point, !last_len
```

⟨*function* Text.start_session 107f⟩≡ (? 0—1)
```
let start_session text =
  text.text_modified
```

⟨*function* Text.commit_session 107g⟩≡ (? 0—1)
```
let commit_session text session_date =
  if text.text_modified > session_date then
    let rec iter session history =
      match history with
        [] -> assert false
      | action :: history ->
          let date =
```

```
            match action with
              Session _ -> failwith "Can not commit nested sessions"
            | Insertion (_,_,date) -> date
            | Deletion (_,_,date) -> date
          in
          if date = session_date then
            text.text_history <- (Session (List.rev (action::session)))
            :: history
          else
            iter (action::session) history
      in
      iter [] text.text_history
```

⟨*function* `Text.with_session` 108⟩≡ (? 0—1)

```
  let with_session f text =
    let session = start_session text in
    let res = f () in
    commit_session text session;
    res
```

Chapter 9

Search and Replace

Search uses the `incremental_mini_bufferX` mechanism: each keystroke in the search minibuffer triggers an immediate search from the saved starting point (`spoint`), moving the main cursor to the match. Pressing `C-s` again within the search advances `spoint` past the current match to find the next occurrence. Replace comes in four flavours: literal or regexp, with or without interactive confirmation (`Query` vs `NoQuery`). The `replace` function wraps all modifications in a `Text.session` so the entire replace operation can be undone as one unit.

9.1 Searching

⟨*searching keys* 109a⟩≡ (25)
```
[ControlMap, Char.code 's'], Search.isearch_forward;
[ControlMap, Char.code 'r'], Search.isearch_backward;
(* standard? *)
[MetaMap, Char.code 's'], Search.isearch_forward_regexp;
[MetaMap, Char.code 'r'], Search.isearch_backward_regexp;
```
Uses `Efuns.mod_ident.MetaMap`.

⟨*type* `Efuns.sens` 109b⟩≡ (? 0—1)
```
type sens =
| Backward
| Forward
```
Uses `Options.value.Value`.

9.1.1 Strings

⟨*constant* `Search.isearch_forward` 109c⟩≡ (? 0—1)
```
let isearch_forward = isearch RegexpString Forward
[@@interactive]
```

⟨*constant* `Search.isearch_backward` 109d⟩≡ (? 0—1)
```
let isearch_backward = isearch RegexpString Backward
[@@interactive]
```

9.1.2 Regexps

⟨*constant* `Search.isearch_forward_regexp` 109e⟩≡ (? 0—1)
```
let isearch_forward_regexp = isearch Regexp Forward
[@@interactive]
```

⟨*constant* `Search.isearch_backward_regexp` 109f⟩≡ (? 0—1)
```
let isearch_backward_regexp = isearch Regexp Backward
[@@interactive]
```

9.1.3 Search mini frame

9.1.4 Secondary cursor

9.1.5 Gap management

OCaml's `Str` regex library operates on flat strings and knows nothing about the gap. Before any search, `clean_text` must move the gap to the end of the buffer (via `move_gpoint_to`) and fill the gap bytes with `'\000'` so that the regex engine sees a contiguous, clean string. The `text_clean` flag avoids repeating this work when multiple searches happen without intervening edits.

⟨`Text.t` *other fields* 110a⟩+≡ (13a) ◁39b 150e▷
```
mutable text_clean : bool;
```

⟨`Text.move_gpoint_to()` *reset* `text_clean` 110b⟩≡ (44b)
```
text.text_clean <- false;
```

⟨*function* `Text.clean_text` 110c⟩≡ (? 0—1)
```
let clean_text text =
  if not text.text_clean then begin
    let size = text.text_size in
    let gsize = text.gsize in
    let string = text.text_string in
    move_gpoint_to text size;
    Bytes.fill string (size - gsize) gsize '\000';
    text.text_clean <- true
  end
```

9.1.6 Helpers

`Search.isearchX` implements incremental search using the `incremental_mini_buffer` mechanism. It saves the starting point in `spoint`, then on every keystroke recompiles the search string into a regex and calls `Text.search_forward` (or backward) from `spoint`. Pressing `C-s`/`C-r` within the search advances `spoint` to the current match and searches again—this is the "repeat" behavior. `C-w` grabs the word at point and appends it to the search string, a feature that makes it easy to search for the identifier under the cursor. If the search string is empty when repeating, `set_last` inserts the previous search.

⟨*constant* `Search.last_search` 110d⟩≡ (? 0—1)
```
let last_search = ref ""
```

⟨*constant* `Search.case_fold` 110e⟩≡ (? 0—1)
```
(* todo: this should be a buffer variable, not a global *)
let case_fold = ref false
```

⟨*function* `Search.isearch` 110f⟩≡ (? 0—1)
```
let isearch to_regexp sens frame =
  let (buf, text, point) = Frame.buf_text_point frame in

  let ismap = Keymap.create () in
  let spoint = Text.dup_point text point in
  let orig = Text.get_position text point in
  ⟨save current pos from frame for position history navigation 100d⟩

  let sens = ref sens in
  let to_regexp = ref to_regexp in

  let request () =
    "isearch-"^
```

```
      (match !sens with
        Backward -> "backward"
      | Forward -> "forward"
    )^
      (match !to_regexp with
        Regexp -> "-regexp :"
      | RegexpString -> " :"
    )
  in

  let string = ref "" in
  let isearch_s () =
    last_search := !string;
    let regexp, _deltaTOREMOVE =
      match !to_regexp, !case_fold with
      | Regexp, true  -> Str.regexp_case_fold !string, 0
      | Regexp, false -> Str.regexp !string, 0
      | RegexpString, true -> Str.regexp_string_case_fold !string, String.length !string
      | RegexpString, false -> Str.regexp_string !string, String.length !string
    in
    Text.goto_point text point spoint;
    (* claude: catch Not_found so failed searches show a message
     * instead of propagating as an uncaught exception *)
    (try
      match !sens with
      | Backward -> Text.search_backward text regexp point |> ignore
      | Forward ->
          let len = Text.search_forward text regexp point in
          Text.fmove text point len
    with Not_found ->
      Message.message frame "Failing I-search"
    )
  in
  let set_last mini_frame =
    if !string = "" then begin
      let (_, text, point) = Frame.buf_text_point mini_frame in
      Text.insert text point !last_search;
      Text.fmove text point (String.length !last_search);
      string := !last_search
    end
  in

  ⟨Search.isearch() key bindings 111⟩
```

⟨Search.isearch() *key bindings* 111⟩≡ (110f)

```
Keymap.add_binding ismap [ControlMap, Char.code 's']
  (fun mini_frame ->
    set_last mini_frame;
    Text.goto_point text spoint point;
    sens := Forward;
    isearch_s ();
    Minibuffer.update_request mini_frame (request ())
);
Keymap.add_binding ismap [ControlMap, Char.code 'r']
  (fun mini_frame ->
    set_last mini_frame;
    Text.goto_point text spoint point;
    sens := Backward;
    isearch_s ();
    Minibuffer.update_request mini_frame (request ())
```

```
  );

  Keymap.add_binding ismap [ControlMap, Char.code 'w']
    (fun mini_frame ->
      (*      set_last mini_frame;       *)
      Text.goto_point text spoint point;
      let end_current_word = Move.end_of_word buf point in
      string := !string ^ end_current_word;
      Edit.insert_string mini_frame end_current_word;
      isearch_s ();
      Minibuffer.update_request mini_frame (request ())
    );

  let _mini_frame =
    Select.incremental_mini_buffer frame ismap (request ()) !string
      (fun _frame str ->
        string := str;
        isearch_s ()
    )
    (fun _frame str ->
        last_search := str;
        Text.remove_point text spoint
    )
  in
  let kill_and f mini_frame =
    last_search := Text.to_string mini_frame.frm_buffer.buf_text;
    Minibuffer.kill mini_frame frame;
    f frame
  in
  Keymap.add_binding ismap [ControlMap, Char.code 'g']
    (fun mini_frame  ->
      Minibuffer.kill mini_frame frame;
      Text.set_position frame.frm_buffer.buf_text frame.frm_point orig
  );
  Keymap.add_binding ismap [NormalMap, XK.xk_Left]
    (fun mini_frame  ->
      Minibuffer.kill mini_frame frame;
      Move.move_backward frame 1 |> ignore
     );
  Keymap.add_binding ismap [NormalMap, XK.xk_Right]
    (fun mini_frame  ->
      Minibuffer.kill mini_frame frame;
      Move.move_forward frame 1 |> ignore
    );
  Keymap.add_binding ismap [NormalMap, XK.xk_Down] (kill_and Move.forward_line);
  Keymap.add_binding ismap [NormalMap, XK.xk_Up] (kill_and Move.backward_line);
  Keymap.add_binding ismap [ControlMap, Char.code 'a']
    (kill_and Move.beginning_of_line);
  Keymap.add_binding ismap [ControlMap, Char.code 'e']
    (kill_and Move.end_of_line)
```

⟨*function* Minibuffer.update_request 112⟩≡ (? 0—1)

```
  let update_request frame request =
    let qlen = String.length request in
    let window = frame.frm_window in
    let top_window = Window.top window in
    let _mini_window = Window.create true window.win_up
        qlen window.win_ypos
        (top_window.top_width - qlen) 1
    in
```

```
    frame.frm_width <- top_window.top_width - qlen;
    frame.frm_xpos <- qlen;
    frame.frm_mini_buffer <- Some request;
    frame.frm_redraw <- true
```

⟨*function* Text.search_forward 113a⟩≡ (? 0—1)

```
  let search_forward text regexp point =
    let gsize = text.gsize in
    let gap_end = text.gpoint.pos + gsize in

    if point.pos = text.gpoint.pos
    then point.pos <- gap_end;
    if point.pos < gap_end
    then clean_text text;

    let gap_end = text.gpoint.pos + gsize in
    let string = Bytes.to_string text.text_string in
    let pos = Str.search_forward regexp string point.pos in
    let pos = if pos >= gap_end then pos - gsize else pos in
    set_position text point pos;
    Str.match_end () - Str.match_beginning ()
```

⟨*function* Text.search_forward_matched 113b⟩≡ (? 0—1)

```
  let search_forward_matched text regexp point =
    let gsize = text.gsize in
    let gap_end = text.gpoint.pos + gsize in

    if point.pos = text.gpoint.pos
    then point.pos <- gap_end;
    if point.pos < gap_end
    then clean_text text;

    let string = Bytes.to_string text.text_string in
    let pos = Str.search_forward regexp string point.pos in
    let pos =
      if pos >= text.gpoint.pos + gsize
      then pos - gsize
      else pos
    in
    set_position text point pos;
    Str.matched_string string
```

⟨*function* Text.search_forward_groups 113c⟩≡ (? 0—1)

```
  let search_forward_groups text regexp point groups =
    let gsize = text.gsize in
    let gap_end = text.gpoint.pos + gsize in

    if point.pos = text.gpoint.pos
    then point.pos <- gap_end;
    if point.pos < gap_end
    then clean_text text;

    let gap_end = text.gpoint.pos + gsize in
    let string = Bytes.to_string text.text_string in
    let pos = Str.search_forward regexp string point.pos in
    let pos = if pos >= gap_end then pos - gsize else pos in
    let array = Array.init groups (fun i -> Str.matched_group (i+1) string) in
    set_position text point pos;
    array
```

⟨*function* Text.search_backward 114a⟩≡ (? 0—1)
```
let search_backward text regexp point =
  if point.pos > text.gpoint.pos
  then clean_text text;
  let string = Bytes.to_string text.text_string in
  let start_pos =
    if point.pos > 0
    then point.pos - 1
    else raise Not_found
  in
  let pos =  Str.search_backward regexp string start_pos in
  set_position text point pos;
  Str.match_end () - Str.match_beginning ()
```

⟨*function* Text.search_backward_groups 114b⟩≡ (? 0—1)
```
let search_backward_groups text regexp point groups =
  if point.pos > text.gpoint.pos
  then clean_text text;
  let string = Bytes.to_string text.text_string in
  let start_pos =
    if point.pos > 0
    then point.pos - 1
    else raise Not_found
  in
  let pos =  Str.search_backward regexp string start_pos in
  let array = Array.init groups (fun i -> Str.matched_group (i+1) string) in
  set_position text point pos;
  array
```

9.2 Replacing

⟨*replacing keys* 114c⟩≡ (25)
```
[MetaMap, Char.code '%'], Search.query_replace_string;
```

⟨*type* Efuns.to_regexp 114d⟩≡ (? 0—1)
```
type to_regexp =
  Regexp
| RegexpString
```

⟨*function* Efuns.string_to_regex 114e⟩≡ (? 0—1)
```
let string_to_regex s = s, Str.regexp s
```

⟨*constant* Efuns.regexp_option 114f⟩≡ (? 0—1)
```
let regexp_option = Options.define_type "Regexp"
    (fun v -> match v with
        Value s ->  string_to_regex s | _ -> raise Not_found)
  (fun (s,_r) -> Value s)
```

9.2.1 Strings

⟨*function* Search.replace_string 114g⟩≡ (? 0—1)
```
let replace_string frame =
  select_replace frame "Replace string: " (fun str ->
  select_replace frame "with string: "    (fun str2 ->
    replace RegexpString frame NoQuery str str2
  ))
[@@interactive]
```

⟨*function* Search.query_replace_string 115a⟩≡ (? 0—1)
```
let query_replace_string frame =
  select_replace frame "Replace string: " (fun str ->
  select_replace frame "with string: "    (fun str2 ->
    replace RegexpString frame (Query ( frame, "Replace string ? (y/n)"))
        str str2
  ))
[@@interactive]
```

9.2.2 Regexps

⟨*function* Search.replace_regexp 115b⟩≡ (? 0—1)
```
let replace_regexp frame =
  select_replace frame "Replace Regexp: " (fun str ->
  select_replace frame "with string: " (fun str2 ->
    replace Regexp frame NoQuery str str2
  ))
[@@interactive]
```

⟨*function* Search.query_replace_regexp 115c⟩≡ (? 0—1)
```
let query_replace_regexp frame =
  select_replace frame "Replace regexp: " (fun str ->
  select_replace frame "with string: "    (fun str2 ->
    replace Regexp frame (Query (frame, "Replace regexp ? (y/n)")) str str2
  ))
[@@interactive]
```

9.2.3 Helpers

The `Search.replaceX` function handles four cases (literal vs. regexp × query vs. no-query) in a single function dispatched by pattern matching. The no-query branches loop with `Text.search_forward`, delete the match, insert the replacement, and advance—all wrapped in a `Text.session` so the entire operation is one undo step. A `last_pos` guard prevents infinite loops when the replacement matches the search pattern (e.g., replacing "a" with "ab"). The query branches use a recursive mini-frame prompting "y/n/!/q" for each match, where "!" switches to no-query mode for the rest.

⟨*function* Text.replace_matched 115d⟩≡ (? 0—1)
```
let replace_matched text repl =
  Str2.replace_matched repl text.text_string
```

⟨*function* Search.select_replace 115e⟩≡ (? 0—1)
```
let select_replace frame request action =
  Select.select_string frame request string_history "" action
```

⟨*function* Search.replace 115f⟩≡ (? 0—1)
```
let replace flag frame query str repl =
  let (_, text, point) = Frame.buf_text_point frame in
  ⟨save current pos from frame for position history navigation 100d⟩
  let regexp = to_regexp flag str in
  let session = Text.start_session text in
  let n = ref 0 in
  try
    let last_pos = ref (Text.get_position text point) in
    match flag,query with
      Regexp, NoQuery ->
        while not (* (Top_window.check_abort frame) *) false do
          let len = Text.search_forward text regexp point in
```

```
      let pos = Text.get_position text point in
      if pos = !last_pos then
        (if Text.fmove_res text point 1 = 0 then raise Exit)
      else begin
          last_pos := pos;
          let result = Text.replace_matched text repl in
          let (_,_matched) = Text.delete_res text point len in
          Text.insert text point result;
          Text.fmove text point (String.length result);
          n := !n + 1
        end
    done
| RegexpString, NoQuery ->
    let delta = String.length repl in
    while not (* (Top_window.check_abort frame) *) false do
      let len = Text.search_forward text regexp point in
      let pos = Text.get_position text point in
      if pos = !last_pos then
        (if Text.fmove_res text point 1 = 0 then raise Exit)
      else begin
          last_pos := pos;
          Text.delete text point len;
          Text.insert text point repl;
          Text.fmove text point delta;
          n := !n + 1
        end
    done
| Regexp, Query (frame,request) ->
    let top_window = Window.top frame.frm_window in
    let rec iter () =
      let len = Text.search_forward text regexp point in
      let pos = Text.get_position text point in
      if pos = !last_pos then begin
          (if Text.fmove_res text point 1 = 0 then raise Exit);
          iter ();
        end
      else begin
          last_pos := pos;
          top_window.top_second_cursor <- Some frame;
          let _ = Select.select_yes_or_no frame request
              (fun yes ->
                if yes then
                  begin
                    let result = Text.replace_matched text repl in
                    let (_,_matched) = Text.delete_res text point len in
                    Text.insert text point result;
                    Text.fmove text point (String.length result);
                    n := !n + 1
                  end;
                iter ()) in
          ()
        end
    in
    iter ()
| RegexpString, Query (frame,request) ->
    let delta = String.length repl in
    let top_window = Window.top frame.frm_window in
    let rec iter () =
      let len = Text.search_forward text regexp point in
      let pos = Text.get_position text point in
```

```
        if pos = !last_pos then begin
            (if Text.fmove_res text point 1 = 0 then raise Exit);
            iter ();
          end
        else begin
            last_pos := pos;
            top_window.top_second_cursor <- Some frame;
            let _ = Select.select_yes_or_no frame request
                (fun yes ->
                  if yes then
                    begin
                      Text.delete text point len;
                      Text.insert text point repl;
                      Text.fmove text point delta;
                      n := !n + 1
                    end;
                  iter ())
            in
            ()
          end
      in
      iter ()
with
| Not_found ->
    Text.commit_session text session;
    Message.message frame ("Replace "^(string_of_int !n)^" occurences")
| _ ->
    Text.commit_session text session
```

Chapter 10

Multiple Buffers

Buffers are identified by unique names (derived from the filename's basename, with a `<N>` suffix for duplicates) and stored in `edt_buffers` and `edt_files` hash tables. Buffer names starting with `*` (like `*scratch*`) are treated as special and excluded from "save modified?" prompts. Killing a buffer decrements its `buf_shared` reference count; the actual cleanup only happens when no frame references it, at which point `Ebuffer.kill` removes the buffer from the global tables and sets `buf_shared` to −1 as a tombstone.

⟨*function* `Simple.buffer_list` 118a⟩≡ (? 0—1)
```
let buffer_list () =
  (Globals.editor()).edt_buffers |> Hashtbl_.to_list |> List.map fst
```

⟨*function* `Ebuffer.get_name` 118b⟩≡ (? 0—1)
```
let get_unique_name filename =
  let basename = Filename.basename filename in
  let name =
    if basename = ""
    then (Filename.basename (Filename.dirname filename)) ^ "/"
    else basename
  in
  let i = ref 0 in
  let compute_name () =
    if !i =|= 0
    then name
    else Printf.sprintf "%s<%d>" name !i
  in
  try
    while true do
      let _ = Hashtbl.find (Globals.editor()).edt_buffers (compute_name ())
      in
      incr i
    done;
    assert false
  with Not_found ->
    compute_name ()
```

10.1 Buffers menu

⟨`Config` *buffers menu setup* 118c⟩≡ (57a)
```
Top_window.buffers_menu := (fun top_window _button () ->
    let buffers = ref [] in
    let edt = Globals.editor() in
    Hashtbl.iter (fun name _buf -> buffers := name :: !buffers) edt.edt_buffers;
    let _desc = Array.map (fun name ->
```

```
        (name, Top_window.wrap top_window (fun top_window ->
              let frame = top_window.top_active_frame in
              let window = frame.frm_window in
              Frame.change_buffer frame.caps window name
          )))
      (Array.of_list !buffers) in

    (* X11 *)
    (*
    let menu = new WX_popup.t top_window.top_root desc in
    let (x,y) = button#root_coordinates in
    menu#popup_once x (y + button#height) (Some !WX_types.button_event)
    *)
    failwith "Std_menu: show menus TODO"
    )
```

Uses `Efuns.editor.edt_buffers`, `Efuns.frame.frm_window` 16a, and `Efuns.top_window.top_active_frame` 66d.

10.2 Changing the buffer

⟨*buffer management keys* 119a⟩≡ (25) 119e▷

```
[c_x; NormalMap, Char.code 'b'], Multi_buffers.change_buffer;
```

Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Complex.change_buffer` 119b⟩≡ (? 0—1)

```
let change_buffer (frame : Frame.t) : unit =
  let default = get_previous_frame () in
  set_previous_frame frame;
  select_buffer frame " Switch to buffer: " default (fun str ->
    Frame.change_buffer frame.caps frame.frm_window str
  )
[@@interactive]
```

⟨*function* `Frame.change_buffer` 119c⟩≡ (? 0—1)

```
let change_buffer (caps : < frame_caps ; ..>) (window : Window.t) (name : string) : unit =
  try
    let buf = Hashtbl.find (Globals.editor()).edt_buffers name in
    let frame = create caps window None buf in
    Hooks.exec_named_hooks !!change_buffer_hooks frame;
    status_name frame buf.buf_name
  with Not_found -> ()
```

10.3 Killing the buffer

Killing a buffer is subtler than it sounds because a buffer is always displayed in at least one frame. `C-x k` first switches the frame to the next buffer (from `prev_buffers` or via `next_buffer`), then checks `buf_shared`: if no other frame references the buffer, `Ebuffer.killX` removes it from the editor's hash tables, runs `buf_finalizers`, and sets `buf_shared` to −1 as a tombstone. If other frames still show the buffer, it stays alive.

⟨*file menu entries* 119d⟩≡ (57c) 137d▷

```
"Kill Buffer", "kill_buffer";
```

⟨*buffer management keys* 119e⟩+≡ (25) ◁119a

```
[c_x; NormalMap, Char.code 'k'], Multi_buffers.kill_buffer;
```

⟨*function* Simple.kill_buffer 120a⟩≡ (? 0—1)

```
let kill_buffer (frame : Frame.t) : unit =
  let window = frame.frm_window in
  let buf = frame.frm_buffer in

  let new_buf =
    match !prev_buffers with
    | x::xs when x <> buf.buf_name ->
        prev_buffers := xs;
        (try
          Hashtbl.find (Globals.editor()).edt_buffers x
        with Not_found ->
          next_buffer buf
        )
    | _ -> next_buffer buf
  in

  let _new_frame = Frame.create frame.caps window None new_buf in
  if buf.buf_shared =|= 0
  then Ebuffer.kill buf
[@@interactive]
```

⟨*function* Ebuffer.kill 120b⟩≡ (? 0—1)

```
let kill buf =
  let edt = Globals.editor() in
  Hashtbl.remove edt.edt_buffers buf.buf_name;
  buf.buf_filename |> Option.iter (fun filename ->
    Hashtbl.remove edt.edt_files filename
  );
  List.iter (fun f -> f () ) buf.buf_finalizers;
(* TODO Gc.compact (); this cause some segfault under plan9 with ocaml light *)
  buf.buf_shared <- -1
```

10.4 Cycling buffers

⟨*constant* Select.prev_buffers 120c⟩≡ (? 0—1)

```
(* todo: maybe this should be in frame.ml, next to change_buffer,
 * so change_buffer can maintain it instead of imposing to everyone
 * to use set_previous_frame just before Frame.change_buffer.
 * todo: or do via change_buffer_hook?
 *)
let prev_buffers = ref []
```

⟨*function* Complex.left_buffer 120d⟩≡ (? 0—1)

```
let left_buffer (frame : Frame.t) : unit =
  set_previous_frame frame;
  Frame.change_buffer frame.caps frame.frm_window
    (match !prev_buffers with
    | name :: buffer :: tail ->
        prev_buffers := tail @ [name];
        buffer
    | _ -> raise Not_found
    )
[@@interactive]
```

⟨*function* Complex.right_buffer 120e⟩≡ (? 0—1)

```
let right_buffer (frame : Frame.t) : unit =
  set_previous_frame frame;
```

```
    Frame.change_buffer frame.caps frame.frm_window
      (match !prev_buffers with
      | name :: tail ->
          (match List.rev tail with
          | buffer :: tail ->
              prev_buffers := name :: (List.rev tail);
              buffer
          | _ -> raise Not_found
          )
      | _ -> raise Not_found
      )
  [@@interactive]
```

⟨*function* `Select.set_previous_frame` 121a⟩≡ (? 0—1)

```
  let set_previous_frame frame =
    let name = frame.frm_buffer.buf_name in
    prev_buffers := name :: (Utils.list_removeq !prev_buffers name)
```

10.5 Pinning buffers

⟨*buffer navigating keys* 121b⟩≡ (25)

```
  (* pinning, lefessant?? *)
  [ControlMetaMap, XK.xk_Down], Multi_buffers.down_buffer;
  [ControlMetaMap, XK.xk_Up], Multi_buffers.up_buffer;
```

Uses `Efuns.mod_ident.ControlMetaMap` 21b and `XK.xk_Up`.

⟨*constant* `Complex.up_buffer` 121c⟩≡ (? 0—1)

```
  let up_buffer = ref ""
```

⟨*function* `Complex.down_buffer` 121d⟩≡ (? 0—1)

```
  let down_buffer frame =
    up_buffer := frame.frm_buffer.buf_name
  [@@interactive]
```

⟨*function* `Complex.up_buffer` 121e⟩≡ (? 0—1)

```
  let up_buffer (frame : Frame.t) : unit =
    if !up_buffer = ""
    then raise Not_found;
    set_previous_frame frame;
    Frame.change_buffer frame.caps frame.frm_window !up_buffer
  [@@interactive]
```

Chapter 11

Multiple Windows

⟨*function* `Window.iter` 122a⟩≡ (? 0—1)
```
let iter f window =
  let rec iter1 window =
    match window.win_down with
    | HComb (w1,w2) -> iter1 w1; iter1 w2
    | VComb (w1,w2) -> iter1 w1; iter1 w2
    | WFrame frame -> f frame
    | NoFrame -> ()
  in
  iter1 window
```

⟨*function* `Frame.active` 122b⟩≡ (? 0—1)
```
let active frame =
  let top_window = Window.top frame.frm_window in
  top_window.top_active_frame <- frame;
  frame.frm_buffer.buf_filename |> Option.iter (fun filename ->
    (Globals.editor()).edt_dirname <- Filename.dirname filename
  )
```

11.1 Creating sub windows

Splitting a window is like `fork`: the current leaf window gets a `VComb` (or `HComb`) with two children, each half the original size. The existing frame is moved into one child via `Frame.installX`; a fresh frame on the same buffer is created in the other. `cut_frameX` is the shared helper that does the geometry: it divides the window height (or width) in two, creates two child windows, wires `win_down`, and returns the empty child for the caller to populate.

⟨`edit_menu` *entries* 122c⟩≡ (62d) 123c▷
```
"Cut Frame", "vertical_cut_frame";
```

⟨*frame management keys* 122d⟩≡ (25) 123d▷
```
[c_x; NormalMap, Char.code '2'], Multi_frames.vertical_cut_frame;
[c_x; NormalMap, Char.code '3'], Multi_frames.horizontal_cut_frame;
```
Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Multi_frames.v_cut_frame` 122e⟩≡ (? 0—1)
```
let vertical_cut_frame frame =
  if frame.frm_mini_buffer = None
  then Frame.create frame.caps (cut_frame frame) None frame.frm_buffer |> ignore
[@@interactive]
```

⟨*function* `Multi_frames.h_cut_frame` 123a⟩≡ (? 0—1)

```
let horizontal_cut_frame (frame : Frame.t) =
  if frame.frm_mini_buffer = None
  then
    let window = frame.frm_window in
    if window.win_width > 10 then begin
      let wi = window.win_width / 2 in
      let w1 = Window.create false
                 (Window window) window.win_xpos window.win_ypos
                 wi window.win_height in
      let w2 = Window.create false (Window window) (window.win_xpos + wi)
                 window.win_ypos
                 (window.win_width - wi) window.win_height in
      window.win_down <- HComb (w1,w2);
      Frame.install w1 frame;
      Frame.create frame.caps w2 None frame.frm_buffer |> ignore
    end
[@@interactive]
```

⟨*function* `Multi_frames.cut_frame` 123b⟩≡ (? 0—1)

```
let cut_frame frame =
  let window = frame.frm_window in
  if window.win_height > 3 then
    let h = window.win_height / 2 in
    let w1 = Window.create false
        (Window window) window.win_xpos window.win_ypos
        window.win_width h in
    let w2 = Window.create false (Window window) window.win_xpos
        (window.win_ypos + h)
      window.win_width (window.win_height - h) in
    window.win_down <- VComb (w1,w2);
    Frame.install w1 frame;
    w2
  else
    window
```

11.2 Deleting sub windows

`C-x 1` (`one_frame`) collapses back to a single frame by re-installing the active frame directly into the top-level window, discarding the split tree. `C-x 0` (`delete_frame`) removes only the current pane by re-installing the active frame into the parent window—effectively replacing the `HComb`/`VComb` with just the surviving child. Both operations call `Frame.killX` on the displaced frames, which decrements their buffers' `buf_shared` counts.

⟨`edit_menu` *entries* 123c⟩+≡ (62d) ◁122c

```
"One Frame", "one_frame";
"Delete Frame", "delete_frame";
```

⟨*frame management keys* 123d⟩+≡ (25) ◁122d

```
[c_x; NormalMap, Char.code '1'], Multi_frames.one_frame;
[c_x; NormalMap, Char.code '0'], Multi_frames.delete_frame;
```

⟨*function* `Multi_frames.one_frame` 123e⟩≡ (? 0—1)

```
let one_frame frame =
  if frame.frm_mini_buffer = None
  then
    let window = frame.frm_window in
    let top_window = Window.top window in
    if not (top_window.window == window) then
```

```
      begin
        Frame.install top_window.window frame;
        Frame.active frame
      end
  [@@interactive]
```

⟨*function* `Multi_frames.delete_frame` 124a⟩≡ (? 0—1)

```
  let delete_frame frame =
    if frame.frm_mini_buffer = None
    then
      let window = frame.frm_window in
      match window.win_up with
      | TopWindow _ -> ()
      | Window upwin ->
          Frame.install upwin frame;
          Frame.active frame
  [@@interactive]
```

⟨*function* `Frame.kill` 124b⟩≡ (? 0—1)

```
  let kill frame =
    let (buf, text, point) = buf_text_point frame in
    ⟨Frame.kill() setting frm_killed field 79e⟩
    buf.buf_shared <- buf.buf_shared - 1;
    Text.remove_point text buf.buf_point;
    Text.remove_point text buf.buf_start;
    buf.buf_point <- point;
    buf.buf_start <- frame.frm_start
```

⟨*function* `Multi_frames.remove_frame` 124c⟩≡ (? 0—1)

```
  let remove_frame frame =
    if frame.frm_mini_buffer = None then
      let window = frame.frm_window in
      match window.win_up with
      | TopWindow _ -> ()
      | Window upwin ->
          Window.prev (Frame.install upwin) window
```

11.3 Navigating windows

⟨*frame navigation keys* 124d⟩≡ (25)

```
  [c_x; NormalMap, Char.code 'o'], Multi_frames.next_frame;
```

⟨*function* `Multi_frames.next_frame` 124e⟩≡ (? 0—1)

```
  let next_frame frame =
    let window = frame.frm_window in
    Window.next Frame.active window
  [@@interactive]
```

⟨*function* `Window.next` 124f⟩≡ (? 0—1)

```
  let rec next f window =
    match window.win_up with
    | TopWindow top_window -> first f top_window.window
    | Window win ->
        match win.win_down with
          HComb (_w1,w2) ->
            if w2 == window
            then next f win
            else first f w2
```

```
      | VComb (_w1,w2) ->
          if w2 == window
          then next f win
          else first f w2
      | _ -> ()
```

⟨*function* `Window.first` 125a⟩≡ (? 0—1)

```
  let rec first f window =
    match window.win_down with
      WFrame frame -> f frame
    | HComb (w1,_w2) -> first f w1
    | VComb (w1,_w2) -> first f w1
    | NoFrame -> ()
```

⟨*function* `Window.last` 125b⟩≡ (? 0—1)

```
  let rec last f window =
    match window.win_down with
      WFrame frame -> f frame
    | HComb (_w1,w2) -> last f w2
    | VComb (_w1,w2) -> last f w2
    | NoFrame -> ()
```

⟨*function* `Window.prev` 125c⟩≡ (? 0—1)

```
  let rec prev f window =
    match window.win_up with
      TopWindow top_window ->
        if window == top_window.window then ()
        else
          last f top_window.window
    | Window win ->
        match win.win_down with
          HComb (w1,_w2) ->
            if w1 == window
            then prev f win
            else last f w1
        | VComb (w1,_w2) ->
            if w1 == window
            then prev f win
            else last f w1
        | _ -> ()
```

Chapter 12

Major Modes

A major mode customizes the editor for a particular file type: it provides a dedicated keymap, syntax coloring, indentation rules, and mode-specific variables. Each buffer has exactly one major mode (stored in `buf_major_mode`), which is auto-detected by matching the filename against `modes_alist`—a list of regexp/mode pairs populated by each mode's start hook. The flow is always the same: an action like "ocaml_mode" calls `Ebuffer.set_major_modeX`, which stores the mode record and executes its `maj_hooks` to install coloring, key bindings, and variables.

⟨`Efuns.buffer` *other fields* 126a⟩+≡ (15a) ◁103b 131a▷
```
  mutable buf_major_mode : major_mode;
```

⟨*type* `Efuns.major_mode` 126b⟩≡ (? 0—1)
```
  and major_mode = {
      maj_name : string;
      maj_map : map;

      mutable maj_hooks : (buffer -> unit) list;
      mutable maj_vars : Store.t;

      (* another customization is the syntax table! don't forget
       * to call Ebuffer.create_syntax_table and modify
       * buf.buf_syntax_table otherwise you'll modify
       * the shared syntax table
       *)
    }
```
Uses `Efuns.buffer`, `Efuns.map`, and `Store.t`.

⟨*function* `Ebuffer.new_major_mode` 126c⟩≡ (? 0—1)
```
  let new_major_mode name hook_opt = {
      maj_name = name;
      maj_map = Keymap.create ();
      maj_hooks = (match hook_opt with None -> [] | Some hook -> [hook]);
      maj_vars = Store.new_store ();
    }
```

⟨*function* `Ebuffer.set_major_mode` 126d⟩≡ (? 0—1)
```
  let set_major_mode buf mode =
    Logs.info (fun m -> m "setting %s major mode" mode.maj_name);
    buf.buf_modified <- buf.buf_modified + 1;
    buf.buf_major_mode <- mode;
    mode.maj_hooks |> List.iter (fun f ->
      try f buf
      with exn -> Error.error_exn "set_major_mode" exn
    )
```

⟨Efuns.status *other fields* 127a⟩+≡ (74b) ◁75e 132a▷
```
mutable stat_mode : major_mode;
```

12.1 Fundamental mode

The fundamental mode is the default major mode—the one every buffer starts with before auto-detection kicks in. Note the French spelling "Fondamental" (the author is French), mirroring Emacs's "Fundamental" mode. The mode record is created with no hooks, meaning it installs no syntax coloring or special key bindings.

⟨*function* Ebuffer.fondamental_mode 127b⟩≡ (? 0—1)
```
let fondamental_mode frame =
  set_major_mode frame.frm_buffer fondamental__mode
[@@interactive]
```

⟨*constant* Ebuffer.fondamental_mode 127c⟩≡ (? 0—1)
```
let fondamental__mode = new_major_mode "Fondamental" None (* no hooks *)
```

12.2 Auto detection

When a buffer is created, the `create_buf_hook` calls `Ebuffer.set_buffer_modeX`, which walks `modes_alist`—a list of (regexp, major_mode) pairs—looking for a match against the buffer's filename. Each language mode registers its own entry at startup (e.g., `"\\.ml$"` for OCaml mode). The regexp list is compiled lazily and cached in `regexp_alist` to avoid recompiling on every buffer creation. The somewhat ugly `raise Exit` pattern is used to short-circuit `List.iter` on the first match, since OCaml's standard library lacks a `List.find_map` equivalent.

⟨*constant* Ebuffer.modes_alist 127d⟩≡ (? 0—1)
```
let modes_alist = Store.create_abstr "modes_alist"
```

⟨*constant* Ebuffer.suffix_reg 127e⟩≡ (? 0—1)
```
let suffix_reg = Str.regexp "\\(.*\\)<[0-9]+>$"
```

⟨*function* Ebuffer.set_buffer_mode 127f⟩≡ (? 0—1)
```
let set_buffer_mode buf =
  let buf_name =
    match buf.buf_filename with
      None ->
        (try
           if Str.string_match suffix_reg buf.buf_name 0
           then Str.matched_group 1 buf.buf_name
           else buf.buf_name
         with exn ->
           Error.error_exn "set_buffer_mode" exn;
           buf.buf_name
        )
    | Some file_name -> file_name
  in
  let modes_alist = Var.get_var buf modes_alist in
  (* must use != here, because modes_alist contain functional values *)
  if (Eq.phys_not_equal !modes_old modes_alist) then begin
    regexp_alist := modes_alist |> List.map (fun (file_reg, major) ->
      Str.regexp file_reg, major
    );
    modes_old := modes_alist;
  end;
  try
```

```
    !regexp_alist |> List.iter (fun (regexp, major) ->
      if Str.string_match regexp buf_name 0
      then
        try
          set_major_mode buf major;
          raise Exit
        with
        | Exit -> raise Exit
        | exn ->
          Error.error_exn "set_buffer_mode" exn;
          raise Exit
    )
  with Exit -> ()
```

⟨*constant* `Ebuffer.modes_old` 128a⟩≡ (? 0—1)

```
let modes_old = ref []
```

⟨*constant* `Ebuffer.regexp_alist` 128b⟩≡ (? 0—1)

```
let regexp_alist = ref []
```

⟨*toplevel* `Ebuffer` *starting hook* 128c⟩≡ (? 0—1)

```
let _ =
  Hooks.add_start_hook (fun () ->
    Var.set_global create_buf_hook [set_buffer_mode];
    Var.set_global modes_alist []
  )
```

12.3 Status line

The status line displays the active major mode name followed by any minor mode names in parentheses, e.g., "(OCaml Abbrev)". `Frame.status_major_modeX` uses physical equality checks (`Eq.phys_equal`) to avoid redrawing the status line when the mode has not actually changed—a small but important optimization since status rendering is called on every display cycle.

⟨`Frame.create_without_top()` *adjust status of frame* 128d⟩+≡ (17b) ◁74f

```
status_major_mode frame;
```

⟨*function* `Frame.status_major_mode` 128e⟩≡ (? 0—1)

```
let status_major_mode frame  =
  let buf = frame.frm_buffer in
  let status = frame.frm_status in
  if not (Eq.phys_equal status.stat_modes buf.buf_minor_modes &&
            Eq.phys_equal status.stat_mode buf.buf_major_mode
    ) then
    begin
      status.stat_modes <- buf.buf_minor_modes;
      status.stat_mode <- buf.buf_major_mode;
      status_print status (Printf.sprintf "(%s)"
          (String.concat " "
            (buf.buf_major_mode.maj_name ::
            (List.map (fun m -> m.min_name) status.stat_modes))))
      StatMode;
    end
```

12.4 Major mode variables and keys

⟨*function* `Var.set_major_var` 128f⟩≡ (? 0—1)

```
let set_major_var maj var value =
  Store.set maj.maj_vars var value
```

⟨*function* `Keymap.add_major_key` 129a⟩≡ (? 0—1)

```
let add_major_key major prefix action =
  add_binding major.maj_map prefix action
```

⟨`Ebuffer.get_binding()` *major mode key search* 129b⟩≡ (23d)

```
(let b = Keymap.get_binding buf.buf_major_mode.maj_map keylist in
  match b with
    Prefix _map -> binding := b
  | Function _f -> binding := b; raise Exit
  | Unbound -> ());
```

⟨`Var.get_var()` *try with major mode variables* 129c⟩≡ (27c)

```
Store.get buf.buf_major_mode.maj_vars var
```

12.5 Customization examples

Beyond variables and keymaps, major modes customize many other aspects of the editing experience. The subsections below sketch the main extension points: syntax coloring (via a lexer), indentation rules, abbreviation tables, code templates ("structures"), compilation error parsing, outline folding, and context-sensitive completion.

12.5.1 Colors

Syntax coloring is driven by the text attribute system described in the Text chapter. Each major mode provides a colorization function (typically an ocamllex-generated lexer) that walks the buffer and sets character attributes for keywords, strings, comments, etc.

12.5.2 Indentation

Indentation is typically bound to `TAB` in each major mode's keymap. The indentation function examines the current and preceding lines to compute the correct indentation level, often using the syntax table to recognize keywords, braces, or other structural markers.

12.5.3 Abbrevs

Abbreviation tables map short strings to their expansions (e.g., "fn" to "function"). When a word-ending character like space is typed, `Abbrevs.expand_sabbrevX` checks whether the word before the cursor matches an entry in the buffer's `abbrev_table` and, if so, replaces it in place.

12.5.4 Structures

Structures are a form of code template: a key sequence inserts a multi-line skeleton containing "holes" marked by consecutive `^^` characters, and `C-c -` (`Simple.next_holeX`) jumps the cursor to the next hole, deleting the marker. This provides a lightweight snippet system without requiring a full template engine.

⟨*misc keys* 129d⟩+≡ (25) ◁84a 140c▷

```
[ c_c; NormalMap, Char.code '-'], Structure.next_hole;
```

Uses `Efuns.mod_ident.NormalMap`.

⟨*function* `Simple.next_hole` 130a⟩≡ (? 0—1)

```
(* a hole is two consecutive '^' chars *)
let next_hole frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.with_dup_point text point (fun curseur ->
    while
      not ((Text.get_char text curseur = '^') && (Text.fmove_res text curseur 1 = 1) &&
        (Text.get_char text curseur = '^')) && (Text.fmove_res text curseur 1 = 1)  do () done;
    if Text.get_char text curseur = '^' then
      (Text.bmove text curseur 1;
        Text.delete text curseur 2;
        Text.goto_point text point curseur);
  )
[@@interactive]
```

⟨*function* `Simple.insert_structure` 130b⟩≡ (? 0—1)

```
let insert_structure s frame =
  let (_, text, point) = Frame.buf_text_point frame in
  Text.insert text point s;
  next_hole frame
```

⟨*function* `Simple.install_structures` 130c⟩≡ (? 0—1)

```
let install_structures buf list =
  list |> List.iter (fun (key, s) ->
    Keymap.add_binding buf.buf_map key (insert_structure s)
  )
```

12.5.5 Compilation and errors

Major modes can customize how compilation errors are parsed by setting the `find_error` variable, which controls how error lines in the `*Compile*` buffer are matched to filenames and line numbers. This is used by `Compil.next_errorX` to jump to the source of each error.

12.5.6 Outline

Outline support lets a major mode define which lines are "headings" (e.g., function definitions, section markers) so the user can fold or navigate the buffer at a structural level. Each mode customizes what counts as a heading, much like `paren_mode` customizes which characters are matchable.

12.5.7 Completion

Major modes can provide context-sensitive completion by setting a completion variable that drives `TAB` or `M-TAB` behavior—for instance, completing identifiers from an index of the current project.

Chapter 13

Minor Modes

Unlike major modes, a buffer can have multiple minor modes active simultaneously (e.g., `paren_mode` for matching-parenthesis highlighting together with `abbrev_mode` for abbreviation expansion). Each minor mode has its own keymap and variables, and the key lookup chain checks all active minor mode maps before falling through to the major mode and global maps. Minor modes are toggled on and off per buffer via `Ebuffer.set_minor_modeX` and `Ebuffer.del_minor_modeX`.

⟨`Efuns.buffer` *other fields* 131a⟩+≡ (15a) ◁126a 136d▷
```
mutable buf_minor_modes : minor_mode list;
```

⟨*type* `Efuns.minor_mode` 131b⟩≡ (? 0—1)
```
and minor_mode = {
    min_name : string;
    min_map : map;

    mutable min_hooks : (buffer -> unit) list;
    mutable min_vars : Store.t;
  }
```

⟨*function* `Ebuffer.new_minor_mode` 131c⟩≡ (? 0—1)
```
let new_minor_mode name hooks  = {
    min_name = name;
    min_map = Keymap.create ();
    min_hooks = hooks;
    min_vars = Store.new_store ()
  }
```

⟨*function* `Ebuffer.set_minor_mode` 131d⟩≡ (? 0—1)
```
let set_minor_mode buf mode =
  buf.buf_minor_modes <- mode :: buf.buf_minor_modes;
  buf.buf_modified <- buf.buf_modified + 1;
  mode.min_hooks |> List.iter (fun f ->
    try f buf
    with exn -> Error.error_exn "set_minor_mode" exn
  )
```

⟨*function* `Ebuffer.del_minor_mode` 131e⟩≡ (? 0—1)
```
let del_minor_mode buf minor =
  buf.buf_minor_modes <-
    List.fold_right (fun mode list ->
      if Eq.phys_equal mode minor then begin
        buf.buf_modified <- buf.buf_modified + 1;
        list
      end else (mode :: list)
    ) buf.buf_minor_modes []
```

⟨Efuns.status *other fields* 132a⟩+≡ (74b) ◁127a
```
mutable stat_modes : minor_mode list;
```
Uses Text.box.

13.1 Example

This section shows the boilerplate for creating a minor mode: allocate a minor_mode record with Ebuffer.new_minor_, add key bindings to its min_map, and register a toggle action so the user can enable or disable the mode with M-x. The sample here is intentionally skeletal—a real minor mode like paren_mode would populate the install hook and key bindings with actual behavior.

⟨*constant* Minor_mode_sample.minor_mode_name 132b⟩≡ (? 0—1)
```
let minor_mode_name = "my_minor"
```

⟨*function* Minor_mode_sample.install 132c⟩≡ (? 0—1)
```
let install _buf =
  ()
```

⟨*constant* Minor_mode_sample.mode 132d⟩≡ (? 0—1)
```
let mode = Ebuffer.new_minor_mode  minor_mode_name [install]
```
Uses Minor_mode_sample.install() 132c and Minor_mode_sample.minor_mode_name 132b.

⟨*function* Minor_mode_sample.minor_mode_fun 132e⟩≡ (? 0—1)
```
let minor_mode_fun _frame = ()
```

⟨*toplevel* Minor_mode_sample._1 132f⟩≡ (? 0—1)
```
let _ =
  List.iter
    (fun key ->
      Keymap.add_binding mode.min_map [NormalMap, Char.code key]
        minor_mode_fun
  )
  [ ] (* no keys *)
```
Uses Efuns.minor_mode.min_map 126b, Efuns.mod_ident.NormalMap, Minor_mode_sample.minor_mode_fun() 132e, and Minor_mode_sample.mode 132d.

⟨*toplevel* Minor_mode_sample._2 132g⟩≡ (? 0—1)
```
let _ =
  Action.define_action (minor_mode_name ^ "_mode")
    (Minor_modes.toggle_minor mode)
```
Uses Minor_mode_sample.minor_mode_name 132b, Minor_mode_sample.mode 132d, and Minor_modes.toggle_minor().

⟨*function* Ebuffer.has_minor_mode 132h⟩≡ (? 0—1)
```
let has_minor_mode buf minor =
  List.memq minor buf.buf_minor_modes
```

13.2 Minor mode variables and keys

⟨*function* Var.set_minor_var 132i⟩≡ (? 0—1)
```
let set_minor_var min var value =
  Store.set min.min_vars var value
```

⟨*function* Keymap.add_minor_key 132j⟩≡ (? 0—1)
```
let add_minor_key minor prefix action =
  add_binding minor.min_map prefix action
```

⟨Ebuffer.get_binding() *minor mode key search* 133a⟩≡ (23d)

```
buf.buf_minor_modes |> List.iter (fun minor ->
  let b = Keymap.get_binding minor.min_map keylist in
  match b with
    Prefix _map -> binding := b
  | Function _f -> binding := b; raise Exit
  | Unbound -> ()
);
```

⟨Var.get_var() *try with minor mode variables* 133b⟩≡ (27c)

```
let rec iter list =
  match list with
  | [] -> raise Not_found
  | min :: list ->
      try
        Store.get min.min_vars var
      with _ -> iter list
in
iter buf.buf_minor_modes
```

Chapter 14

Extra Features

This chapter covers the features that make efuns a practical development environment rather than just a text editor: running shell commands and compilation inside buffers, a client-server protocol for opening files from the command line, dynamic abbreviation expansion, text transformations, and various smaller conveniences.

14.1 Running external commands

`M-!` runs a shell command and captures its output in a `*Command*` buffer, embodying the Emacs philosophy that "everything is a buffer"—the output can then be searched, copied, or further manipulated with all the usual editing commands. Under the hood, `System.systemX` forks a child process with redirected pipes and uses `Concur.add_readerX` to poll the output asynchronously, feeding it into a `Text.t` buffer. Unlike GNU Emacs (which is single-threaded and must use non-blocking I/O or sub-processes), efuns can use OCaml threads, making the implementation more straightforward.

⟨*external commands keys* 134a⟩≡ (25)
```
[MetaMap, Char.code '!'], System.shell_command;
```

⟨*constant* `System.shell_hist` 134b⟩≡ (? 0—1)
```
let shell_hist = ref []
```

⟨*function* `System.shell_command` 134c⟩≡ (? 0—1)
```
let shell_command (frm : Frame.t) =
  Select.select_string frm "Run command:" shell_hist "" (fun cmd ->
    let pwd = (Globals.editor()).edt_dirname in
    start_command frm.caps pwd "*Command*" (Multi_frames.cut_frame frm) cmd None |> ignore)
[@@interactive]
```

⟨*function* `System.start_command` 134d⟩≡ (? 0—1)
```
let start_command (caps : < caps; Cap.env; .. >) (pwd : string) (buf_name : string)
  window cmd end_action_opt =
  let end_action =
    match end_action_opt with
    | None   -> (fun _buf _status -> ())
    | Some f -> f
  in
  let buf = system caps pwd buf_name cmd end_action in
  let frame = Frame.create caps window None buf in
  frame
```

⟨*function* System.system 135⟩≡ (? 0—1)

```
let system (caps : < caps; .. >) (pwd : string) (buf_name : string) (cmd : string) end_action =
  let (pid,inc,outc) = open_process caps pwd cmd in
  let text = Text.create "" in
  let curseur = Text.new_point text in
  let buf = Ebuffer.create buf_name None text (Keymap.create ()) in
  (* !!! *)
  buf.buf_sync <- true;

  let ins = Unix.descr_of_in_channel inc in
  let tampon = Bytes.create 1000 in
  let active = ref true in
  let edt = Globals.editor () in
  Concur.add_reader ins (fun () ->
    let _pos,str = Text.delete_res text curseur
                   (Text.point_to_eof text curseur) in
    let len = input inc tampon 0 1000 in
    Mutex.lock edt.edt_mutex;
    if len =|= 0 then begin
      let _pid,status = CapUnix.waitpid caps [Unix.WNOHANG] pid in
      (match status with
      | Unix.WEXITED s ->
          Text.insert_at_end text (spf "Exited with status %d\n" s);
          close_in inc;
          close_out outc;
          (try end_action buf s with _ -> ())
      | _ -> Text.insert_at_end text "Broken pipe"
      );
      Text.set_position text curseur (Text.size text);
      active := false;
      (* redraw screen *)
      Top_window.update_display ();

      Mutex.unlock edt.edt_mutex;
      Concur.remove_reader ins; (* Kill self *)
    end
    else
      Text.insert_at_end text (Bytes.sub_string tampon 0 len);

    Text.set_position text curseur (Text.size text);
    Text.insert text curseur str;
    buf.buf_modified <- buf.buf_modified +1;

    (* redraw screen *)
    Top_window.update_display ();
    Mutex.unlock edt.edt_mutex
  );

  let lmap = buf.buf_map in
  Keymap.add_binding lmap [NormalMap, XK.xk_Return] (fun frame ->
    let point = frame.frm_point in
    Text.insert text point "\n";
    Text.fmove text point 1;
    if !active then (* to avoid a segmentation fault in Ocaml *) begin
      let str = Text.sub text curseur
          (Text.point_to_eof text curseur) in
      Text.set_position text curseur (Text.size text);
      (* synchronize viewpoint *)
      output_substring outc str 0 (String.length str);
      flush outc
```

```
    end
  );
  ⟨System.system() set finalizer, to intercept killed frame 137a⟩
  buf
```

⟨*function* `System.open_process` 136a⟩≡ (? 0—1)

```
let open_process (caps : < caps; .. >) (pwd : string) (cmd : string) : Proc.pid * in_channel * out_channel =
  let (in_read, in_write) = Unix.pipe() in
  let (out_read, out_write) = Unix.pipe() in
  let inchan = Unix.in_channel_of_descr in_read in
  let outchan = Unix.out_channel_of_descr out_write in
  match CapUnix.fork caps () with
  | 0 ->
      if out_read <> Unix.stdin then begin
        Unix.dup2 out_read Unix.stdin;
        Unix.close out_read
      end;
      if in_write <> Unix.stdout ||  in_write <> Unix.stderr then begin
        if in_write <> Unix.stdout
        then Unix.dup2 in_write Unix.stdout;
        if in_write <> Unix.stderr
        then Unix.dup2 in_write Unix.stderr;
        Unix.close in_write
      end;
      List.iter Unix.close [in_read;out_write];
      (* I prefer to do it here than in the caller *)
      CapSys.chdir caps pwd;
      CapUnix.execv caps "/bin/sh" [| "/bin/sh"; "-c"; cmd |];
      (* never here! *)
      (* nosemgrep: do-not-use-exit *)
      exit 127 (* for ocaml light *)
  | pid ->
      Unix.close out_read;
      Unix.close in_write;
      (pid, inchan, outchan)
```

⟨*function* `Text.insert_at_end` 136b⟩≡ (? 0—1)

```
let insert_at_end text str =
  low_insert text text.text_size str |> ignore;
  text.text_history <- [];
  text.text_modified <- text.text_modified + 1
```

14.1.1 Threads and mutex

Because external commands run in separate threads that write into shared `Text.t` buffers, efuns needs a global mutex (`edt_mutex`) to protect buffer mutations from data races. The reader callback in `System.systemX` acquires this lock before inserting output and releases it after triggering a display update.

⟨`Efuns.editor` *other fields* 136c⟩+≡ (20a) ◁52c

```
edt_mutex : Mutex.t;
```

14.1.2 Finalizers

When a buffer attached to a running process is killed (e.g., the user closes the `*Command*` window), the child process must be cleaned up. The `buf_finalizers` list holds cleanup closures that are called when the buffer is destroyed—here, sending `SIGKILL` to the subprocess and removing the reader from the event loop.

⟨`Efuns.buffer` *other fields* 136d⟩+≡ (15a) ◁131a 137b▷

```
mutable buf_finalizers : (unit -> unit) list;
```

⟨System.system() *set finalizer, to intercept killed frame* 137a⟩≡ (135)
```
buf.buf_finalizers <- (fun () ->
  (try
     CapUnix.kill caps pid Sys.sigkill;
     CapUnix.waitpid caps [] pid |> ignore;
   with _ -> ()
  );
  Concur.remove_reader ins
) :: buf.buf_finalizers;
```

14.1.3 Synchronous buffers

A synchronous buffer (`buf_sync = true`) automatically scrolls to the end of the text on every display update, so the user always sees the latest output—essential for command and compilation buffers where new lines arrive asynchronously from a child process. Without this flag, the frame would stay at whatever scroll position the user last had.

⟨Efuns.buffer *other fields* 137b⟩+≡ (15a) ◁136d
```
mutable buf_sync : bool;
```

⟨Frame.display() *if buf sync goto end of text* 137c⟩≡ (64a)
```
if buf.buf_sync && buf.buf_modified <> frame.frm_last_buf_updated
then Text.set_position text point (Text.size text);
```

14.2 Compilation and grep mode

`M-x compile` and `M-x grep` both reuse the same infrastructure as `M-!`: they spawn a child process and capture output in a buffer (`*Compile*` or `*Grep*`). The key addition is an `end_action` callback that colorizes the output buffer when the process finishes, and the `compilation_frame` global that remembers the compilation window so that successive compilations reuse the same frame rather than splitting new ones.

⟨*file menu entries* 137d⟩+≡ (57c) ◁119d
```
"Compile", "compile";
```

14.2.1 M-x grep

`Compil.grepX` prompts for a grep command, spawns it, and stores the resulting frame and an `error_point` in `compilation_frame` so that `C-x ` ` can step through the matches. If a previous compilation buffer exists, it is killed and its window reused—avoiding an ever-growing stack of split frames.

⟨*constant* Compil.grep_command 137e⟩≡ (? 0—1)
```
let grep_command = define_option ["compil"; "grep_command"] "" string_option
    "grep -n"
```

⟨*constant* Compil.grep_hist 137f⟩≡ (? 0—1)
```
let grep_hist = ref ["grep -n "]
```

⟨*function* Compil.grep 138a⟩≡ (? 0—1)
```
let grep frame =
  let default = List.hd !grep_hist in
  Select.select_string frame
    (Printf.sprintf "Grep command: %s (default: %s) " !!grep_command default)
    grep_hist ""
    (fun cmd ->
      let cmd = if cmd = "" then default else cmd in
      let cmd = !!grep_command ^ " " ^ cmd in
      let cdir = Frame.current_dir frame in
      let comp_window =
        match !compilation_frame with
        | None -> Multi_frames.cut_frame frame
        | Some (new_frame,error_point, _) ->
            Text.remove_point new_frame.frm_buffer.buf_text error_point;
            Ebuffer.kill new_frame.frm_buffer;
            if new_frame.frm_killed
            then Multi_frames.cut_frame frame
            else new_frame.frm_window
      in
      let comp_frame = System.start_command frame.caps cdir "*Grep*" comp_window cmd
         (Some (fun buf _status -> color_buffer buf))
      in
      Frame.active frame;
      let buf = comp_frame.frm_buffer in
      let error_point = Text.new_point buf.buf_text in
      compilation_frame := Some (comp_frame, error_point, cdir)
      (* todo? no set major mode? *)
  )
[@@interactive]
```

⟨*function* Frame.current_dir 138b⟩≡ (? 0—1)
```
let current_dir frame =
  let buf = frame.frm_buffer in
  match buf.buf_filename with
  | Some filename -> Filename.dirname filename ^ "/"
  | None -> (Globals.editor()).edt_dirname ^ "/"
```

14.2.2 M-x compile

`Compil.compileX` works like grep but with an extra trick: if the compile command starts with "make", it walks up the directory tree looking for a `Makefile` so the build runs from the project root rather than the current file's directory. After spawning the process, it propagates error-parsing variables (`find_error`, `find_error_location_regexp`) from the source buffer to the compilation buffer, and sets a dedicated compilation major mode.

⟨*constant* Compil.make_hist 138c⟩≡ (? 0—1)
```
let make_hist = ref [!!make_command]
```

⟨*constant* Compil.make_command 138d⟩≡ (? 0—1)
```
(* old: was "make -k" to keep going after first error, but I prefer without*)
let make_command = define_option ["compil";"make_command"] ""
    string_option "make"
```

⟨*function* Compil.compile 138e⟩≡ (? 0—1)
```
let compile (frame : Frame.t) =
  let default = List.hd !make_hist in
  Select.select_string frame
```

```
      ("Compile command: (default :"^ default^") " )
      make_hist ""
      (fun cmd ->
        let cmd = if cmd = "" then default else cmd in (* cmd ||| default *)
        let cdir = Frame.current_dir frame in
        ⟨Compil.compile() find possibly cdir with a makefile 140b⟩
        let comp_window =
          match !compilation_frame with
          | None -> Multi_frames.cut_frame frame
          | Some (new_frame,error_point, _) ->
              Text.remove_point new_frame.frm_buffer.buf_text error_point;
              Ebuffer.kill new_frame.frm_buffer;
              if new_frame.frm_killed
              then Multi_frames.cut_frame frame
              else new_frame.frm_window
        in
        (* this makes also comp_frame the active frame *)
        let comp_frame =
          System.start_command frame.caps cdir "*Compile*" comp_window cmd
          (Some (fun buf _status -> color_buffer buf))
        in
        (* switch back cursor to original frame *)
        Frame.active frame;
        let buf = comp_frame.frm_buffer in
        let error_point = Text.new_point buf.buf_text in
        compilation_frame := Some (comp_frame, error_point, cdir);

        (* propagate vars *)
        let buf2 = frame.frm_buffer in
        (try
           let x = Var.get_var buf2 find_error in
           Var.set_local buf find_error x
         with Not_found | Failure _ -> ()
        );
        (try
           let x =  Var.get_var buf2 find_error_location_regexp in
           Var.set_local buf find_error_location_regexp x
         with Not_found | Failure _ -> ()
        );
        (try
           let x =  Var.get_var buf2 find_error_error_regexp in
           Var.set_local buf find_error_error_regexp x
         with Not_found | Failure _ -> ()
        );
        (* set major mode *)
        Ebuffer.set_major_mode buf mode

    )
  [@@interactive]
```

⟨*constant* `Compil.compilation_frame` 139⟩≡ (? 0—1)

```
  let compilation_frame = ref None
```

14.2.3 Finding the makefile automatically

The Makefile search walks from the current directory upward via `Filename.dirname` until it finds a `Makefile`, `makefile`, or `GNUmakefile`. If none is found (i.e., `dirname` returns the same directory), it falls back to the

original directory. This is controlled by the `compile_find_makefile` option, which defaults to `true`.

⟨*constant* `Compil.compile_find_makefile` 140a⟩≡ (? 0—1)
```
let compile_find_makefile = define_option ["compil";"find_makefile"] ""
    bool_option true
```

⟨`Compil.compile()` *find possibly cdir with a makefile* 140b⟩≡ (138e)
```
let cdir, cmd =
  match () with
  | _ when cmd =~ "^cd +\\([^;]+\\);\\(.*\\)$" ->
      let (dir, cmd) = Regexp_.matched2 cmd in
      let finaldir = expand_tilde frame.caps dir in
      finaldir, cmd
  | _ when !!compile_find_makefile && cmd =~ "^make" ->
    (* try to find a Makefile in the directory *)
      let rec iter dir =
        if Sys.file_exists (Filename.concat dir "Makefile") ||
           Sys.file_exists (Filename.concat dir "makefile") ||
           Sys.file_exists (Filename.concat dir "GNUmakefile")
        then dir
        else
          let newdir = Filename.dirname dir in
          if newdir = dir
          then cdir
          else iter newdir
      in
      iter cdir, cmd
  | _ -> cdir, cmd
in
```

14.2.4 Go to next error

`C-x ``` (`Compil.next_errorX`) advances the `error_point` through the compilation buffer, calling the mode-specific `find_error` parser to extract the filename, line, and column. It then opens (or finds) the error file in a separate frame and positions the cursor at the error location. If the compilation frame was killed, it is "unkilled" into a new split window so the user can continue stepping through errors.

⟨*misc keys* 140c⟩+≡ (25) ◁129d 144a▷
```
[c_x;NormalMap, Char.code '`' ], Compil.next_error;
```
Uses `Efuns.mod_ident.NormalMap`.

⟨*constant* `Compil.find_error` 140d⟩≡ (? 0—1)
```
let find_error = Store.create_abstr "find_error"
```

⟨*function* `Compil.next_error` 140e⟩≡ (? 0—1)
```
let next_error (top_frame : Frame.t) : unit =
  match !compilation_frame with
  | None -> Message.message top_frame "No compilation started"
  | Some (frame, error_point, cdir) ->
      if frame.frm_killed
      then Frame.unkill (Multi_frames.cut_frame top_frame) frame;

      let (buf, text, point) = Frame.buf_text_point frame in
      let find_error = find_error_of_buf buf in
      try
        let error = find_error text error_point in
        Text.set_position text frame.frm_start error.err_msg;
        Text.set_position text point error.err_msg;
        frame.frm_force_start <- true;
```

```
        frame.frm_redraw <- true;
        if error.err_filename <> "" then
          let filename =
            if error.err_filename =~ "^/"
            then error.err_filename
            else Filename.concat cdir error.err_filename
          in
          let buf = Ebuffer.read filename (Keymap.create ()) in
          (* new frame for error buffer *)
          let frame =
            try Frame.find_buffer_frame buf
            with Not_found ->
                if Eq.phys_equal frame top_frame then
                  let new_window = Top_window.create top_frame.caps
                      (*Window.display top_window*)
                  in
                  Frame.create frame.caps new_window.window None buf
                else
                  Frame.create frame.caps top_frame.frm_window None buf
          in
          let text = buf.buf_text in
          let point = frame.frm_point in
          Text.point_to_line text point error.err_line;
          Text.fmove text point error.err_begin;
          Frame.active frame
      with Not_found ->
        Message.message top_frame "No more errors"
  [@@interactive]
```

14.3 Efuns server and client

14.3.1 The server

⟨*constant* Server.user 141a⟩≡ (? 0—1)

```
  let user (caps : < Cap.env; .. >) =
    try CapSys.getenv caps "USER"
    with _ -> "noname"
```

⟨*constant* Server.socket_name 141b⟩≡ (? 0—1)

```
  let socket_name (caps : < Cap.env; .. >) =
    (Printf.sprintf "/tmp/efuns-server.%s.%s:0" (user caps) !Globals.displayname)
```

⟨*constant* Server.started 141c⟩≡ (? 0—1)

```
  let started = ref false
```

⟨*type* Server.proto 141d⟩≡ (? 0—1)

```
  type command =
    | LoadFile of string (* filename *) * int (* pos *) * int (* line *) * string
```

⟨*function* Server.read_command 141e⟩≡ (? 0—1)

```
  let read_command (caps : < frame_caps; .. >) (fd : Unix.file_descr) : unit =
    let inc = in_channel_of_descr fd in
    try
      let cmd = input_value inc in
      match cmd with
      | LoadFile (name, pos, line, str) ->
          let window =
              let edt = Globals.editor () in
```

```
            match edt.top_windows with
            | [] -> failwith "no top windows"
            | x::_ -> x.top_active_frame.frm_window
        in
        let frame = Frame.load_file caps window name in
        let (_, text, point) = Frame.buf_text_point frame in
        if pos <> 0
        then Text.set_position text point pos;
        if line <> 0
        then Text.goto_line text point line;
        if str <> ""
        (* bugfix: need the begin/end here, otherwise update below in then *)
        then begin
          let regexp = Str.regexp_string str in
          Text.search_forward text regexp point |> ignore;
        end;
        Top_window.update_display ()
    with
    | End_of_file ->
       Concur.remove_reader fd
    | e ->
      Logs.err (fun m -> m "server read_command exn: %s (thread %d)"
            (Printexc.to_string e) (Thread.id (Thread.self ())));
      Concur.remove_reader fd
```

⟨*function* Server.module_accept 142a⟩≡ (? 0—1)

```
  let module_accept (caps : < frame_caps; ..>) s =
    let fd,_ = accept s in
    Unix.set_close_on_exec fd;
    Concur.add_reader fd (fun _ -> read_command caps fd)
```

⟨*function* Server.start 142b⟩≡ (? 0—1)

```
  (* old: this used to take an optional frame but was useless I think *)
  let start (caps : < frame_caps; ..>) =
    if not !started then
    Utils.catchexn "Efuns server:" (fun _ ->
        let s = Unix.socket PF_UNIX SOCK_STREAM 0 in
        let name = socket_name caps in
        if Sys.file_exists name
        then begin
          Logs.warn (fun m -> m "socket file %s already exists; cancelling the server"
                name);
          (* alt: unlink here, so if you run multiple efuns, the last one wins*)
        end else begin
          Unix.bind s (ADDR_UNIX name);
          Unix.listen s 254;
          Unix.set_nonblock s;
          Unix.set_close_on_exec s;
          Logs.info (fun m -> m "starting server listening on %s (thread %d)"
                     name (Thread.id (Thread.self ())));
          Concur.add_reader s (fun _ ->
            started := true;
            module_accept caps s
          );
         Hooks.add_hook Misc_features.exit_hooks (fun () -> Unix.unlink name);
        end
    )
```

14.3.2 The client: `efuns_client`

The client is a small standalone program that connects to the server socket, marshals a `LoadFile` command with the filename, position, line, and search string, and exits. It converts relative paths to absolute before sending, since the server may be running in a different working directory.

⟨*constant* `Efuns_client.pos` 143a⟩≡ (? 0—1)
```
let pos = ref 0
```

⟨*constant* `Efuns_client.string` 143b⟩≡ (? 0—1)
```
let string = ref ""
```

⟨*constant* `Efuns_client.filename` 143c⟩≡ (? 0—1)
```
let filename = ref None
```

⟨*toplevel* `Efuns_client._1` 143d⟩≡ (? 0—1)
```
let _ = Arg.parse [
    "-pos", Arg.Int (fun i -> pos := i)," <int>: set position";
    "-line", Arg.Int (fun i -> line := i)," <int>: set line";
    "-str", Arg.String (fun s -> string := s)," <string>: search string";
  ] (fun name -> filename := Some name)
  "efuns_server: connect to efuns"
```

⟨*constant* `Efuns_client.pos` *(*`(client/efuns_client.ml)`*)* 143e⟩≡ (? 0—1)
```
let pos = !pos
```

⟨*constant* `Efuns_client.filename` *(*`(client/efuns_client.ml)`*)* 143f⟩≡ (? 0—1)
```
let filename = match !filename with
    None -> failwith "efuns_server: filename missing"
  | Some s -> s
```

⟨*constant* `Efuns_client.string` *(*`(client/efuns_client.ml)`*)* 143g⟩≡ (? 0—1)
```
let string = !string
```

⟨*constant* `Efuns_client.filename` *(*`client/efuns_client.ml`*)2* 143h⟩≡ (? 0—1)
```
let filename =
  if Filename.is_relative filename
  then Filename.concat (Sys.getcwd ()) filename
  else filename
```

⟨*constant* `Efuns_client.display` 143i⟩≡ (? 0—1)
```
let display =
  ""
  (*Xlib.openDisplay "" *)
```

⟨*constant* `Efuns_client.socket_name` 143j⟩≡ (? 0—1)
```
let user =
  try Sys.getenv "USER"
  with _ -> "noname"

let socket_name =
  (Printf.sprintf "/tmp/efuns-server.%s.%s:0" user display)
  (* (Xlib.getWholeProperty display root atom).gp_value *)
```

⟨*toplevel* `Efuns_client._2` 143k⟩≡ (? 0—1)
```
let _ =  if not (Sys.file_exists socket_name) then raise Not_found
```

⟨*toplevel* `Efuns_client._3` 143l⟩≡ (? 0—1)
```
let (_inc,outc) = Unix.open_connection (Unix.ADDR_UNIX socket_name)
let _ =
  output_value outc (Server.LoadFile (filename,pos,!line, string));
  flush outc
```

14.4 M-x eval

This section would allow the user to evaluate arbitrary OCaml expressions at runtime, much like Emacs's `M-x eval-expression` for Emacs Lisp. The implementation is currently commented out because it requires OCaml's dynamic linking / toplevel evaluation facilities, which add significant complexity.

14.5 Completion

⟨*misc keys* 144a⟩+≡ (25) ◁140c 166b▷

```
[NormalMap, Char.code ' '], Abbrevs.char_expand_abbrev;
[MetaMap, Char.code '/'], Abbrevs.dabbrev_expand;
```

Uses `Efuns.mod_ident.NormalMap` and `XK.xk_Insert`.

⟨*function* `Abbrevs.expand_sabbrev` 144b⟩≡ (? 0—1)

```
let expand_sabbrev frame =
  try
    let (buf, text, point) = Frame.buf_text_point frame in
    let abbrevs = Var.get_local buf abbrev_table in
    let str =
      Text.with_dup_point text point (fun mark ->
        Move.to_begin_of_word text point buf.buf_syntax_table;
        Text.region text point mark
      )
    in
    let len = String.length str in
    try
      let repl = Hashtbl.find abbrevs str in
      Text.delete text point len;
      Text.insert text point repl;
      Text.fmove text point (String.length repl); ()
    with
      Not_found -> Text.fmove text point len; ()
  with _ -> ()
```

⟨*function* `Abbrevs.dabbrev_expand` 144c⟩≡ (? 0—1)

```
let dabbrev_expand frame =
  let (buf, text, point) = Frame.buf_text_point frame in
  let syntax = buf.buf_syntax_table in
  let loop = ref false in
  let s, buf, pos, history =
    match !dabbrev_buf with
      Some (s, for_frame, for_position, for_len,buf, pos, history)
      when frame == for_frame && for_position = Text.get_position text point ->
        Text.bmove text point for_len;
        Text.delete text point for_len;
        s, buf, pos, history
    | _ ->
        let str = Move.beginning_of_word buf point in
        str , buf,
        Text.get_position text point - String.length str, []
  in
  let truelen = String.length s in
  let s = escaped s in
  Text.bmove text point truelen;
  let regexp = Str.regexp s in
  let rec iter curr_buf pos =
    let curr_text = curr_buf.buf_text in
    Text.with_new_point curr_text (fun mark ->
```

```
      Text.set_position curr_text mark pos;
      try
        let rec restart () =
          let _len = Text.search_backward curr_text regexp mark in
          if curr_text == text && mark = point then
            raise Exit
          else
            if Text.bmove_res curr_text mark 1 <> 0 then
              if syntax.(Char.code (Text.get_char curr_text mark))
              then begin Text.fmove curr_text mark 1; restart () end
              else Text.fmove curr_text mark 1
            else
              ()
        in
        let rec first () =
          restart ();
          Text.fmove curr_text mark truelen;
          let m = Move.end_of_word curr_buf mark in
          if List.mem m history then begin
            Text.bmove curr_text mark truelen;
            first ()
          end else begin
            Text.fmove text point truelen;
            let _,len = Text.insert_res text point m in
            Text.fmove text point len;
            dabbrev_buf := Some (s, frame, Text.get_position text point, len,
              curr_buf, Text.get_position curr_text mark, m :: history);
          end
        in
        first ()
      with
      | Not_found ->
          let curr_buf = Multi_buffers.next_buffer curr_buf in
          if curr_buf == buf
          then
            if !loop
            then raise Not_found
            else loop := true (* to avoid infinite loop *)
          ;
          iter curr_buf (Text.size curr_buf.buf_text)
      | Exit ->
          Text.fmove text point truelen;
          dabbrev_buf := None;
          raise Not_found
    )
  in
  iter buf pos; ()
[@@interactive]
```

⟨*function* `Simple.next_buffer` 145⟩≡ (? 0—1)

```
(* useful when want to iterate over all buffers, e.g. in dabbrev_expand *)
let next_buffer buf =
  let buf_list = Utils.list_of_hash (Globals.editor()).edt_buffers in
  let rec iter list =
    match list with
      [] -> raise Not_found
    | (_name,b) :: tail ->
        if Eq.phys_equal b buf
        then
          match tail with
```

```
          | [] -> snd (List.hd buf_list) (* go back to head *)
          | (_,b)::_ -> b
        else
          iter tail
  in
  iter buf_list
```

⟨*constant* `Abbrevs.dabbrev_buf` 146a⟩≡ (? 0—1)

```
let dabbrev_buf = ref None
```

⟨*constant* `Abbrevs.abbrev_table` 146b⟩≡ (? 0—1)

```
let abbrev_table = Store.create "abbrev_table"
    (fun table ->
      let s = ref "" in
      Hashtbl.iter (fun s1 s2 ->
          s := Printf.sprintf "%s%s --> %s\n" !s s1 s2
      ) table;
      !s
  )
  Store.no_input
```

⟨*function* `Abbrevs.escaped` 146c⟩≡ (? 0—1)

```
let escaped s =
  let n = ref 0 in
  let len = String.length s in
  for i = 0 to len - 1 do
    let c = String.get s i in
    match c with
      '\\' | '[' | ']' | '^' -> incr n
    | _ -> ()
  done;
  if !n = 0 then s else
  let ss = Bytes.create (len + !n) in
  let p = ref 0 in
  for i = 0 to len - 1 do
    let c = String.get s i in
    (match c with
        '\\' | '[' | ']' | '^' ->
          Bytes.set ss !p '\\';
          incr p
      | _ -> ());
    Bytes.set ss !p c;
    incr p
  done;
  Bytes.to_string ss
```

14.6 Colored filenames

This section would colorize filenames in directory listings using the same color scheme as `ls --color` (driven by `dircolors.ml`), so that executables, directories, symlinks, etc. are visually distinguished in file-selection buffers.

14.7 Transforming

Text transformations operate on characters, words, or paragraphs. `C-t` transposes two characters, `M-t` transposes two words, `M-l`/`M-u`/`M-c` change word case, and `M-q` fills a paragraph to 75 columns. These all use

`Text.with_session` so the entire transformation can be undone in a single step.

⟨*transforming keys* 147a⟩≡ (25)
```
[ControlMap, Char.code 't'], (Frame.to_frame Edit.transpose_chars);
[MetaMap, Char.code 't'], (Frame.to_frame Edit.transpose_words);

[MetaMap, Char.code 'l'], (fun frm ->
  Edit.on_word frm.frm_buffer frm.frm_point String.lowercase_ascii
);
[MetaMap, Char.code 'u'], (fun frm ->
  Edit.on_word frm.frm_buffer frm.frm_point String.uppercase_ascii
);
[MetaMap, Char.code 'c'], (fun frm ->
  Edit.on_word frm.frm_buffer frm.frm_point String.capitalize_ascii
);
[MetaMap, XK.xk_q], Transform.fill_paragraph;
```
Uses `Efuns.frame.frm_buffer`, `Efuns.frame.frm_point`, `Efuns.mod_ident.MetaMap`, and `XK.xk_q`.

14.7.1 Characters

`Edit.transpose_charsX` swaps the character before the cursor with the one after it and advances the cursor, so repeated invocations "bubble" a character forward through the line.

⟨*function* `Simple.transpose_chars` 147b⟩≡ (? 0—1)
```
let transpose_chars buf point =
  let text = buf.buf_text in
  text |> Text.with_session (fun () ->
    Text.bmove text point 1;
    let _pos, c1 = Text.delete_res text point 1 in
    Text.fmove text point 1;
    Text.insert text point c1;
  );
  Text.fmove text point 1;
  ()
```

14.7.2 Words

Word transposition is more involved: it must find the boundaries of two adjacent words using the syntax table, extract both, delete them, and re-insert them in swapped order. `Edit.on_wordX` is a helper that extracts the word under the cursor, applies a string function (e.g., `String.lowercase_ascii`), and replaces the original in place.

⟨*function* `Simple.transpose_words` 147c⟩≡ (? 0—1)
```
let transpose_words buf point =
  let text = buf.buf_text in
  text |> Text.with_session (fun () ->
    let syntax = buf.buf_syntax_table in
    in_prev_word text point syntax;
    to_begin_of_word text point syntax;
    Text.with_dup_point text point (fun pos1 ->
      to_end_of_word text point syntax;
      let _,word1 = Text.delete_res text pos1 (Text.distance text pos1 point) in
      Text.goto_point text point pos1;
      in_next_word text point syntax;
      Text.with_dup_point text point (fun pos2 ->
        to_end_of_word text point syntax;
        let _,word2 = Text.delete_res text pos2 (Text.distance text pos2 point) in
        Text.insert text pos1 word2;
```

```
        Text.insert text pos2 word1;
        Text.fmove text point (String.length word1);
    )))
```

⟨*function* `Simple.on_word` 148a⟩≡ (? 0—1)

```
  let on_word buf point f =
    let text = buf.buf_text in
    text |> Text.with_session (fun () ->
      let syntax = buf.buf_syntax_table in
      to_begin_of_word text point syntax;
      Text.with_dup_point text point (fun pos1 ->
        to_end_of_word text point syntax;
        let _,word1 = Text.delete_res text pos1 (Text.distance text pos1 point) in
        let w = f word1 in
        Text.insert text pos1 w;
        Text.fmove text point (String.length w);
      )
    )
```

14.7.3 Filling

`M-q` (`Transform.fill_paragraphX`) reflows a paragraph to fit within 75 columns. It first calls `Simple.simplifyX` to collapse all whitespace (spaces, tabs, newlines) into single spaces, then walks the resulting single line and inserts newlines at the last space before column 75. The `line_comment` variable would eventually allow filling to preserve comment prefixes, but this is not yet implemented.

⟨*constant* `Simple.line_comment` 148b⟩≡ (? 0—1)

```
  let line_comment = Store.create_abstr "Fill_mode.line_comment"
```

⟨*function* `Simple.fill_paragraph` 148c⟩≡ (? 0—1)

```
  (* We will have to modify this to handle line_comment soon !! *)
  let fill_paragraph frame =
    let (buf, text, point) = Frame.buf_text_point frame in
    text |> Text.with_session (fun () ->
      Text.with_dup_point text point (fun start ->
      Move.backward_paragraph buf start;
      Text.with_dup_point text start (fun fin ->
        Move.forward_paragraph buf fin;

        simplify text start fin;
        Text.insert text start "\n";
        let rec iter count last_space =
          if Text.compare text start fin < 0 then
          if Text.fmove_res text start 1 =|= 1 then
            let c = Text.get_char text start in
              if c =$= ' ' then (* good, a new space *)
                iter (count+1) 0
            else
            if count > 75 && count <> last_space then
                begin
                Text.bmove text start (last_space+1);
                Text.delete text start 1;
                Text.insert text start "\n";
                Text.fmove text start 1;
                iter 0 0
                end
              else
                iter (count+1) (last_space+1)
      in
```

```
      iter 0 0;
      Text.insert text fin "\n";
  )))
[@@interactive]
```

⟨*function* `Simple.simplify` 149a⟩≡ (? 0—1)

```
let simplify text start point =
  Text.with_dup_point text start (fun start ->
    let rec iter last_c =
      if start < point then
        let c = Text.get_char text start in
        if c =$= ' ' || c =$= '\n' || c =$= '\t' then
          ( Text.delete text start 1;
            iter ' ')
        else
        if last_c =$= ' ' then
          ( Text.insert text start " ";
            Text.fmove text start 2;
            iter 'a')
        else
          ( Text.fmove text start 1;
            iter 'a')
    in
    iter 'a'
  )
```

14.8 Misc

14.8.1 Electric

An electric command is one that performs extra processing beyond simple character insertion. Here, `electric_insert` inserts a space and then checks whether the line exceeds 75 columns; if so, it searches backward for a word boundary and inserts a newline there, providing automatic line wrapping as the user types—useful in fill-mode or when editing comments.

⟨*function* `Simple.electric_insert_space` 149b⟩≡ (? 0—1)

```
let electric_insert_space frame =
  Edit.self_insert_command frame;
  let (buf, text, point) = Frame.buf_text_point frame in
  let line_len = Text.point_to_bol text point in
  if line_len > 75 then
    Text.with_dup_point text point (fun mark ->
      try
        while (Move.backward_word buf mark;
            Text.point_to_bol text mark > 75) do () done;
        Move.forward_word buf mark;
        Move.backward_word buf mark;
        Text.insert text mark "\n"
      with Not_found -> ()
    )
```

14.8.2 Hungry

`C-Backspace` (`Electric.hungry_electric_deleteX`) deletes backward through contiguous whitespace in one shot, collapsing it to a single space. This is handy for reformatting code where indentation has gotten out of

hand—one keypress removes an entire run of spaces, tabs, and newlines.

⟨*deleting keys* 150a⟩+≡ (25) ◁101g
```
(* ?? *)
[ControlMap, XK.xk_BackSpace], Electric.hungry_electric_delete;
```

⟨*function* `Simple.hungry_char` 150b⟩≡ (? 0—1)
```
let hungry_char c =
  c = ' ' || c = '\n' || c = '\t'
```

⟨*function* `Simple.hungry_electric_delete` 150c⟩≡ (? 0—1)
```
let hungry_electric_delete frame =
  let (_, text, _) = Frame.buf_text_point frame in
  text |> Text.with_session (fun () ->
   let c1 = previous_char frame in
   Edit.delete_backspace_char frame;
   let c2 = previous_char frame in
   if hungry_char c1 && hungry_char c2 then
     try
       Edit.delete_backspace_char frame;
       while
         let c = previous_char frame in
         hungry_char c
       do
         Edit.delete_backspace_char frame
       done;
       Edit.insert_char frame ' '
     with Not_found -> ()
  )
[@@interactive]
```

⟨*function* `Simple.previous_char` 150d⟩≡ (? 0—1)
```
let previous_char frame =
  let (_, text, point) = Frame.buf_text_point frame in
  if Text.bmove_res text point 1 = 0 then raise Not_found;
  let c = Text.get_char text point in
  Text.fmove text point 1;
  c
```

14.8.3 Read-only

⟨`Text.t` *other fields* 150e⟩+≡ (13a) ◁110a
```
mutable text_readonly : bool;
```

⟨`Text.low_insert()` *fail if readonly buffer* 150f⟩≡ (46b 43e)
```
if text.text_readonly
then failwith "Buffer is read-only";
```

⟨*function* `Text.readonly` 150g⟩≡ (? 0—1)
```
let readonly text =
  text.text_readonly
```

⟨*function* `Text.toggle_readonly` 150h⟩≡ (? 0—1)
```
let toggle_readonly text =
  text.text_readonly <- not text.text_readonly
```

Chapter 15

Advanced Topics

This chapter collects features that add complexity but are not central to understanding the editor's architecture: tab handling (variable-width character representation that complicates the box model), Unicode and accent support, signal handling, and dynamic linking of mode plugins.

15.1 Tabs

Tab characters have variable-width display: a tab at column 3 renders as 6 spaces (to reach the next multiple of 9), while a tab at column 8 renders as a single space. This is handled by the `tabreprs` array, which is indexed by `repr_cursor mod 9` to select the right-width string. The author considers this unnecessary complexity—it complicates the box model and rendering code throughout the editor.

⟨`Ebuffer.create()` *adjust charreprs* 151a⟩+≡ (15f) ◁53e
```
  buf.buf_charreprs.(9) <- String.make !tab_size ' ';
```

⟨*constant* `Ebuffer.tab_size` 151b⟩≡ (? 0—1)
```
  let tab_size = ref 9
```

⟨`Text.compute_representation()` *compute charrepr, special char* 151c⟩≡ (54c)
```
  if charcode = 9
  then tabreprs.(!repr_cursor mod 9)
```

⟨`Text.compute_representation()` *compute* `char_repr`*, special char* 151d⟩≡ (54c)
```
  if char_code = 9
  then tabreprs.(!repr_cursor mod 9)
```

⟨*constant* `Text.tabreprs` 151e⟩≡ (? 0—1)
```
  let tabreprs = [|
      "         ";
      "        ";
      "       ";
      "      ";
      "     ";
      "    ";
      "   ";
      "  ";
      " ";
      ""
    |]
```

⟨*constant* `Minibuffer.charreprs` 151f⟩≡ (? 0—1)
```
  let charreprs = Array.init 256 (fun i -> String.make 1 (Char.chr i))
```

⟨*toplevel* `Minibuffer._1` 152a⟩≡ (? 0—1)
```
let _ =
  charreprs.(9) <- String.make !Ebuffer.tab_size ' '
```

15.2 Unicode

Unicode support would require replacing the byte-indexed `Text.t` with a representation that handles multi-byte characters, and extending the 256-entry `charreprs` array to support arbitrary code points. This section is currently a placeholder.

15.3 Accents and AZERTY

French AZERTY keyboards produce accented characters (e, e, c, etc.) via dedicated keys. `Edit.char_insert_commar` wraps the insertion of a specific character, bypassing the normal `self_insert_command` path which uses the X11 key event's character code. The actual key mappings are left as an empty list here and would be populated with the appropriate X11 keysyms in a startup hook.

⟨*function* `Simple.char_insert_command` 152b⟩≡ (? 0—1)
```
let char_insert_command char frame =
  let buf = frame.frm_buffer in
  ⟨Simple.self_insert_command() if overwrite mode 167a⟩
  else insert_char frame char
```

⟨`core_map` *entries* 152c⟩+≡ (83b) ◁83e
```
([
(* see pbs_accent.txt in this directory
*)
] |> List.map (fun (key, char) ->
    (* special for AZERTY keyboards *)
        [NormalMap, key], (Edit.char_insert_command char)
    )
) @
```
Uses `Efuns.mod_ident.NormalMap`.

15.4 Mini maps

15.5 Signals

⟨*exception* `Main.SigInt` 152d⟩≡ (? 0—1)
```
exception SigInt
```

⟨`main()` *set signal handlers* 152e⟩≡ (29c)
```
Utils.register_exn (fun e ->
  match e with
  | Unix.Unix_error (error,f,arg) ->
      Printf.sprintf "Unix error %s: %s %s" f (Unix.error_message error) arg
  | _ -> raise e
);
(* TODO: not in ocaml-light: Printexc.record_backtrace true; *)

Sys.signal Sys.sigint (Sys.Signal_handle (fun _ ->
  (* Should auto-save all buffers, and then exit ... *)
  (* old: was exit 1 but not sure why, SigInt seems more logical *)
```

```
    raise SigInt
  )) |> ignore;
  Sys.signal Sys.sigterm (Sys.Signal_handle (fun _ ->
    (* Should auto-save all buffers, and then exit ... *)
    raise SigInt
  )) |> ignore;
  Sys.signal Sys.sighup (Sys.Signal_handle (fun _ ->
    raise SigInt
  )) |> ignore;
```

15.6 Dynamic linking

⟨main() *command line options* 153a⟩+≡ (29c) ◁31e 157b▷

```
    "-I",Arg.String (fun s -> Globals.load_path =:=
        (Utils.string_to_path s) @ !!Globals.load_path), "<path>: Load Path";
```

⟨*constant* Efuns.load_path 153b⟩≡ (? 0—1)

```
  let load_path = define_option ["efuns_path"]
    "<load_path> is the path where modules (.cmo and .cma) can be found
    for dynamic linking." path_option []
```

⟨*constant* Efuns.path 153c⟩≡ (? 0—1)

```
  let path = (*Dyneval.load_path*) ref []
```

⟨*constant* Efuns.efuns_path 153d⟩≡ (? 0—1)

```
  let efuns_path = [
        (Filename.concat Utils.homedir ".efuns") ;
  (*
        Version.efuns_lib;
        Version.installdir;
        Version.ocamllib
  *)
    ]
```

⟨*toplevel* Efuns *load path* 153e⟩≡ (? 0—1)

```
  let _ =
    path := !!load_path @ efuns_path;
    option_hook load_path (fun _ -> path := !!load_path @ efuns_path)
```

15.7 Misc

This section collects commented-out or vestigial code for X11 resource handling, multiple top-level windows (what Emacs calls “frames”), and other features that are not currently active.

Chapter 16

Conclusion

You now know how efuns works: a gap buffer stores the text, a hierarchy of frames, windows, and top windows provides the display, and an event-driven loop dispatches keys through cascading keymaps to action functions. The redisplay engine automatically repaints only what changed, freeing actions from having to manage the screen. The architecture is remarkably faithful to GNU Emacs's design—buffers, frames, modes, the minibuffer, the kill ring—but written in OCaml rather than C + Emacs Lisp. OCaml's type system catches many errors at compile time that would be runtime surprises in Emacs Lisp, and features like pattern matching and algebraic types make the code more concise (compare the `window_down` variant type to Emacs's C struct gymnastics). On the other hand, the lack of a built-in extension language means users must recompile to add features, unless dynamic linking is used.

Appendix A

Error Management

A.1 Logging

⟨*function* `Efuns.error` 155a⟩≡ (? 0—1)
```
  let error_exn s exn =
    let bt = Printexc.get_backtrace () in
    (* less: this introduces a dependency to pfff, but allows for
     * better reporting. An alternative solution would be to register
     * exn handlers/formatters
     *)
    (match exn with
  (* TODO: to restore, but then depends on semgrep lib_parsing
    | Parsing_error.Lexical_error _ | Parsing_error.Syntax_error _ ->
       let ex = Exception.catch exn in
       (* TODO
       let err = Error_code.exception_to_error "XXX" ex in
       pr2 (Error_code.string_of_error err)
        *)
       UCommon.pr2 (Dumper.dump ex)
  *)
    | _ -> ()
    );
    Logs.err (fun m -> m "error: %s (exn = %s). backtrace:\n%s"
                         s (Printexc.to_string exn) bt)
```

A.2 *Messages*

⟨*function* `Ebuffer.message` 155b⟩≡ (? 0—1)
```
  (* todo: vs Message.message? *)
  let message _buf m =
    let name = "*Messages*" in
    try
      let buf = Hashtbl.find (Globals.editor()).edt_buffers name in
      Text.insert_at_end buf.buf_text (m^"\n");
    with Not_found ->
      create name None (Text.create (m^"\n")) (Keymap.create ()) |> ignore
```

A.3 *backtrace*

⟨`Top_window.handle_key()` *handle exception of* `try_map` 155c⟩+≡ (35d) ◁82b
```
  | e ->
```

```
(* bugfix: call get_backtrace() first! otherwise the code above
 * may throw exn, which can be catched, which modifies the backtrace
 *)
let bt = Printexc.get_backtrace () in
let str = spf "Uncaught exception %s" (Utils.printexn e) in
Logs.err (fun m -> m "%s" str);
message top_window  str;
let buf = Ebuffer.default "*backtrace*" in
let text = buf.buf_text in
Text.insert_at_end text str;
Text.insert_at_end text "\n";
Text.insert_at_end text bt;
Text.insert_at_end text "\n"
```

Appendix B

Debugging

⟨*constants* `Efuns.debug_xxx` 157a⟩≡ (? 0—1)
```
let debug = ref false
(* alt: use Logs src like in semgrep but ocaml-light does not support
 * functors so simpler to use those extra globals to mimic Logs src
 *)
let debug_graphics = ref false
```

⟨`main()` *command line options* 157b⟩+≡ (29c) ◁153a
```
"-verbose", Arg.Unit (fun () -> level := Some Logs.Info),
" verbose mode";
"-debug", Arg.Unit (fun () ->
   Globals.debug := true;
   level := Some Logs.Debug;
),
" debug mode";
"-quiet", Arg.Unit (fun () -> level := None),
" no logs";
"-debug_graphics", Arg.Unit (fun () ->
      level := Some Logs.Debug;
      Globals.debug_graphics := true),
" for debugging";
```

B.1 Actions trace: `-debug`

⟨`Top_window.try_map()` *if debug, print action name* 157c⟩≡ (81c)
```
if !Globals.debug
then begin
  let found = ref false in
  Action.actions |> Hashtbl.iter  (fun k v ->
    match v with
    (* subtle: this will work only if f2 was not a closure *)
    | FrameAction f2 when Eq.phys_equal f f2 ->
        found := true;
        Logs.debug (fun m -> m "action: %s" k)
    | _ -> ()
  );
  if not !found
  then Logs.err (fun m -> m "action not found");
end;
```

B.2 Graphics commands trace: `-debug_graphics`

Appendix C

Profiling

Appendix D

Heterogeneous `Store`

D.1 Interface

⟨*signature type* `Store.t` 159a⟩≡ (? 0—1)
```
type t
```

⟨*signature function* `Store.new_store` 159b⟩≡ (? 0—1)
```
(* constructor *)
val new_store : unit -> t
```

⟨*signature type* `Store.var` 159c⟩≡ (? 0—1)
```
type 'a var
```

⟨*signature function* `Store.get` 159d⟩≡ (? 0—1)
```
val get : t -> 'a var -> 'a
```

⟨*signature function* `Store.set` 159e⟩≡ (? 0—1)
```
val set : t -> 'a var -> 'a -> unit
```

⟨*signature functions* `Store.create_xxx` 159f⟩≡ (? 0—1) 160g▷
```
val create_int : string -> int var
val create_string : string -> string var
val create_float : string -> float var
```

D.2 Implementation

D.2.1 `vars`

⟨*type* `Store.t` 159g⟩≡ (? 0—1)
```
type t = (string, Obj.t ref) Vars.t ref
```

⟨*module Store.Vars* 159h⟩≡ (? 0—1)
```
module Vars = Map_
```

⟨*function* `Store.new_store` 159i⟩≡ (? 0—1)
```
let new_store () =
  ref Vars.empty
```

D.2.2 `var`

⟨*type* `Store.var` 159j⟩≡ (? 0—1)
```
type 'a var = string
```

⟨*function* `Store.create_int` 160a⟩≡ (? 0—1)
```
let create_int name =
  create name string_of_int int_of_string
```

⟨*function* `Store.create_float` 160b⟩≡ (? 0—1)
```
let create_float name =
  create name string_of_float float_of_string
```

⟨*function* `Store.create_string` 160c⟩≡ (? 0—1)
```
let create_string name =
  create name id id
```

⟨*global* `Store.vars_table` 160d⟩≡ (? 0—1)
```
let (vars_table :
    (string, (Obj.t -> string) * (string -> Obj.t)) Hashtbl.t)
  = Hashtbl.create 203
```

⟨*signature function* `Store.create` 160e⟩≡ (? 0—1)
```
val create : string -> ('a -> string) -> (string -> 'a) -> 'a var
```

⟨*function* `Store.create` 160f⟩≡ (? 0—1)
```
let create name print input =
  try
    let _ = Hashtbl.find vars_table name in
    failwith (Printf.sprintf "A variable named %s already exists" name)
  with
    Not_found ->
      Hashtbl.add vars_table name (Obj.magic print, Obj.magic input);
      name
```

⟨*signature functions* `Store.create_xxx` 160g⟩+≡ (? 0—1) ◁159f
```
val create_abstr : string -> 'a var
```

⟨*function* `Store.create_abstr` 160h⟩≡ (? 0—1)
```
let create_abstr name =
  create name no_print no_input
```

⟨*function* `Store.no_print` 160i⟩≡ (? 0—1)
```
let no_print x =
  "<abstr>:" ^ (Dumper.dump x)
```

⟨*function* `Store.no_input` 160j⟩≡ (? 0—1)
```
let no_input (_s : string) = failwith "This variable can not be set"
```

D.2.3 Getter/setter

⟨*function* `Store.get` 160k⟩≡ (? 0—1)
```
let get vars var =
  Obj.magic !(Vars.find var !vars)
```

⟨*function* `Store.set` 160l⟩≡ (? 0—1)
```
let set vars var value =
  let value = Obj.repr value in
  try
    let r = Vars.find var !vars in
    r := value
  with
    Not_found ->
      vars := Vars.add var (ref value) !vars
```

D.2.4 Misc

⟨*signature functions* `Store` 161a⟩≡ (? 0—1)
```
val no_input : string -> 'a
val no_print : 'a -> string

val set_input : t -> 'a var -> string -> unit
val get_print : t -> 'a var -> string
val list : t -> string list

val print : t -> string -> string
val input : t -> string -> string -> unit
```

⟨*function* `Store.get_print` 161b⟩≡ (? 0—1)
```
let get_print vars var =
  let value = get vars var in
  let (p, _i) = Hashtbl.find vars_table var in
  p value
```

⟨*function* `Store.set_input` 161c⟩≡ (? 0—1)
```
let set_input vars var value =
  let (_p, i) = Hashtbl.find vars_table var in
  set vars var (i value)
```

⟨*function* `Store.list` 161d⟩≡ (? 0—1)
```
let list vars =
  let list = ref [] in
  Vars.iter (fun var _value ->
      list := var :: !list;
  ) !vars;
  !list
```

⟨*constant* `Store.print` 161e⟩≡ (? 0—1)
```
let print = get_print
```

⟨*constant* `Store.input` 161f⟩≡ (? 0—1)
```
let input = set_input
```

Appendix E

Configuration File

E.1 Options

E.2 Parameter

⟨Config._5 *start hooks options* 162a⟩≡ (32e)
```
Parameter.add_option_parameter Compil.compile_find_makefile;
Parameter.add_option_parameter Text.add_amount;
```
Uses Default_config.init_global_map().

⟨*type* Simple.parameter 162b⟩≡ (? 0—1)
```
type t =
 string * ((string -> Obj.t) * (Obj.t -> string) * Obj.t Options.t)
```

⟨*constant* Simple.parameters_var 162c⟩≡ (? 0—1)
```
let parameters_var = Store.create_abstr "parameters"
```

⟨*function* Simple.add_option_parameter 162d⟩≡ (? 0—1)
```
let add_option_parameter option =
  add_parameter (shortname option)
   (fun s -> from_value (get_type option) (Value s))
   (fun v ->
      match to_value (get_type option) v with
        Value s -> s
      | _ -> failwith "Unable to print option"
   )
   option
```

⟨*function* Simple.add_parameter 162e⟩≡ (? 0—1)
```
let add_parameter (name : string) (input : string -> 'a)
  (print : 'a -> string) (param : 'a Options.t) =
  let (input : string -> Obj.t) = Obj.magic input in
  let (print : Obj.t -> string) = Obj.magic print in
  let (param : Obj.t Options.t) = Obj.magic param in
  Var.set_global parameters_var (
    (name, (input, print, param)) ::
    (try Var.get_global parameters_var with _ -> []))
```

⟨*constant* Simple.all_params 162f⟩≡ (? 0—1)
```
let all_params = ref None
```

⟨*function* Simple.all_parameters 163a⟩≡ (? 0—1)

```
let all_parameters _frame _ =
  let parameters =
    try Var.get_global parameters_var with _ -> []
  in
  match !all_params with
    Some (f,l) when f == parameters -> l
  | _ ->
      let list = List.map fst parameters in
      all_params := Some (parameters, list);
      list
```

⟨*constant* Complex.parameters_hist 163b⟩≡ (? 0—1)

```
let parameters_hist = ref []
```

⟨*function* Complex.set_parameter 163c⟩≡ (? 0—1)

```
let set_parameter frame =
  let parameters = Var.get_global Parameter.parameters_var in
  Select.select frame "set-parameter : " parameters_hist
    "" (Parameter.all_parameters frame) (fun s -> s) (fun variable ->
      Select.select_string frame (Printf.sprintf "%s : " variable)
      value_hist "" (fun value ->
          let (input,_print,param) = List.assoc variable parameters
          in
          param =:= input value))
[@@interactive]
```

⟨*function* Complex.get_parameter 163d⟩≡ (? 0—1)

```
let get_parameter frame =
  let parameters = Var.get_global Parameter.parameters_var in
  Select.select frame "get-parameter : " parameters_hist
    "" (Parameter.all_parameters frame) (fun s -> s) (fun variable ->
      Top_window.mini_message frame
        (Printf.sprintf "%s : %s" variable (
          let (_input,print,param) = List.assoc variable parameters
          in
          print !!param)))
[@@interactive]
```

Appendix F

Major Mode Examples

F.1 Shell mode

F.2 Buffers menu

F.3 Dired

F.4 Lisp mode

⟨*function* `Text.lexing` 164⟩≡ (? 0—1)

```
let lexing text curseur end_point =
  clean_text text;
  Lexing.from_function (fun str len ->
    let len = min len (distance text curseur end_point) in
    let len = blit str text curseur len in
    fmove text curseur len;
    len
  )
```

Appendix G

Minor Mode Examples

G.1 Abbrevs

G.2 Paren

⟨*constant* `Paren_mode.mode` 165a⟩≡ (? 0—1)
```
let mode = Ebuffer.new_minor_mode "paren" []
```

⟨*function* `Paren_mode.find_matching` 165b⟩≡ (? 0—1)
```
let find_matching frame =
  Edit.self_insert_command frame;
  highlight_paren frame
```

⟨*toplevel* `Paren_mode._1` 165c⟩≡ (? 0—1)
```
let _ =
  (* alt: we could use a major mode var that specifies what is a parenthesis *)
  [')'; '}'; ']'] |> List.iter (fun key ->
    Keymap.add_minor_key mode [NormalMap, Char.code key] find_matching
  )
```

⟨*function* `Paren_mode.paren_mode` 165d⟩≡ (? 0—1)
```
let paren_mode =
  Minor_modes.toggle_minor mode
[@@interactive]
```

⟨*function* `Simple.is_paren_end` 165e⟩≡ (? 0—1)
```
let is_paren_end c = (c == '}') || (c == ']') || (c == ')')
```
⟨`Simple.is_paren_end()` *extra conditions* ??⟩

⟨*function* `Simple.is_paren_begin` 165f⟩≡ (? 0—1)
```
let is_paren_begin c = (c == '{') || (c == '[') || (c == '(')
```
⟨`Simple.is_paren_begin()` *extra conditions* ??⟩

⟨*function* `Simple.highlight_paren` 165g⟩≡ (? 0—1)
```
let highlight_paren frame =
  let (buf, text, point) = Frame.buf_text_point frame in

  (* bugfix: I was using Text.with_dup_point but we must store cursor
   * in highlighted_chars, which then was causing some out_of_bound exn
   * when exciting efuns, and also some weird display bugs,
   * because remove_point was putting some -1 in cursor!
```
⟨`Simple.highlight_paren()` *special code for HTML modes* ??⟩

```
 *)
let cursor = Text.dup_point text point in
try
  (* no previous char (this should never happen when called from
   * find_matching) *)
  if Text.bmove_res text cursor 1 = 0
  then raise Exit;

  let c = Text.get_char text cursor in
  (* right now we highlight only when you type a closing paren *)
  if not (is_paren_end c)
  then raise Exit;

  let rec iter stack =
    if Text.bmove_res text cursor 1 = 0
    then begin
      Top_window.mini_message frame "No matching parenthesis";
      raise Exit
    end;

    let d = Text.get_char text cursor in
    if is_paren_end d
    then iter (d :: stack)
    else
      if is_paren_begin d
      then
        match stack with
        | [] -> (* found matching par *)
          let attr = Text.get_attr text cursor in
          ⟨Simple.highlight_paren() remember highlighted chars 51e⟩
          Text.set_attr text cursor (attr lor Text.highlight_bit);
          buf.buf_modified <- buf.buf_modified + 1
        | _c :: stack -> (* don't try to match *)
          (* we could check they are matching chars (like '{' with '}') *)
          iter stack
      else iter stack
  in
  iter []
with Exit ->
 Text.remove_point text cursor
```

G.3 Fill

G.4 Overwrite mode

⟨*constant* `Simple.overwrite_mode` 166a⟩≡ (? 0—1)

```
let overwrite_mode = Ebuffer.new_minor_mode "Over" []
```

⟨*misc keys* 166b⟩+≡ (25) ◁144a

```
[NormalMap, XK.xk_Insert], Edit.toggle_overwrite_mode;
```

⟨*function* `Edit.toggle_overwrite_mode` 166c⟩≡ (? 0—1)

```
let toggle_overwrite_mode frm =
  let buf = frm.frm_buffer in
  let mode = overwrite_mode in
  if Ebuffer.has_minor_mode buf mode
  then Ebuffer.del_minor_mode buf mode
```

```
    else Ebuffer.set_minor_mode buf mode
  [@@interactive "overwrite_mode"]
```

⟨Simple.self_insert_command() *if overwrite mode* 167a⟩≡ (152b 83d)

```
  if Ebuffer.has_minor_mode buf overwrite_mode
  then insert_at_place frame char
```

⟨*function* Simple.insert_at_place 167b⟩≡ (? 0—1)

```
  let insert_at_place frame char =
    let (_, text, point) = Frame.buf_text_point frame in
    let c = Text.get_char text point in
    if c = '\n'
    then insert_char frame char
    else begin
      Text.delete text point 1;
      Bytes.set single_char 0 char;
      Text.insert text point (Bytes.to_string single_char);
      Text.fmove text point 1
    end
```

Appendix H

Extra Code

Glossary

```
BOL  = Beginning Of Line
EOL  = End Of Line
BOF  = Beginning Of File
EOF  = End Of File
LOF  = ??
LOL  = ??
```

Indexes

Here is a list of the identifiers used, and where they appear. Underlined entries indicate the place of definition. This index is generated automatically.

Bibliography

[Knu92] Donald E. Knuth. *Literate Programming*. Center for the Study of Language and Information, 1992. cited page(s) 9

[Pad09] Yoann Padioleau. Syncweb, literate programming meets unison, 2009. https://github.com/aryx/syncweb. cited page(s) 9