Emacsy: An Embeddable Emacs-like Library for GNU Guile Scheme

Shane Celis
shane.celis@gmail.com

Contents

Preface

This project is an experiment, actually two experiments. Firstly, it's an experiment to see whether there's any interest and utility in an embeddable Emacs-like environment. Secondly, I'd like to see how literate programming fares in comparison to the conventional approach. Let me elaborate a little on each.

Emacs is the extensible programmer's text editor. For decades, it's gobbled up functionality that sometimes seems far removed from text editing. I will expand upon why I believe this is the case and what particular functionality I hope to replicate later. I'd like to discuss a little about why I'm bothering to start with Emacs rather than just writing something entirely new. Emacs has fostered a community of people that are comfortable using, customising, and extending Emacs while its running. The last part is most important in my mind. Extending Emacs is a natural part of its use; it's a tinkerer's dream toy. And I want to grease the rails for people who already *get* what kind of tool I'm trying to provide. Had I chosen another perfectly competent language like Lua instead of a Lisp, that would erect a barrier to that track. Were I to write a completely different API, that's yet another barrier. Were I to "modernize" the terminology used by Emacs, e.g., say "key shortcut" instead of "key binding", or "window" instead of "frame", that's a barrier to drawing the community of people that already *get it* to try this out.

Let me say a little about why I'm choosing to do this as a literate program. I've written a lot of code, none of which was written literately. Recently I had an experience that made me want to try something different. I began a group project. There wasn't *that* much code. Yet not too far into the project, it had become opaque to one of the original contributors. This was a small codebase with someone who was there from the start, and already we were having problems. Maybe the code was bad. Maybe we were bad programmers (Eek!). Whatever the case, assuming there's no simple fix for opaque code, it is something that can be addressed. Better communication about the code may help. So I would like to invest a good faith effort in attempting to write this program in a literate fashion.

A few notes on my personal goals for this document and the code. The writing style I'm leaving as informal for purposes of expediency and lowering the barrier of contribution. Also for expediency, my initial interest is in fleshing out the functionality. I'm not concerned about optimality of the implementation yet. Only in cases where the design cannot be reimplemented to be more efficient would I be concerned. If we can make a useable system, optimization will follow and hopefully be informed by profiling.

There's a ton of work left to do! Please feel free to contribute to the effort.

TODO List

Part I

Usage

Chapter 1

Introduction

Emacsy is inspired by the Emacs text editor, but it is not an attempt to create another text editor. This project "extracts" the kernel of Emacs that makes it so extensible. There's a joke that Emacs is a great operating system—lacking only a decent editor. Emacsy is the Emacs OS sans the text editor. Although Emacsy shares no code with Emacs, it does share a vision. This project is aimed at Emacs users and software developers.

1.0.1 Vision

Emacs has been extended to do much more than text editing. It can get your email, run a chat client, do video editing[1], and more. For some the prospect of chatting from within one's text editor sounds weird. Why would anyone want to do that? Because Emacs gives them so much control. Frustrated by a particular piece of functionality? Disable it. Unhappy with some unintuitive key binding? Change it. Unimpressed by built-in functionality? Rewrite it. And you can do all that while Emacs is running. You don't have to exit and recompile.

The purpose of Emacsy is to bring the Emacs way of doing things to other applications natively. In my mind, I imagine Emacs consuming applications from the outside, while Emacsy combines with applications from the inside—thereby allowing an application to be Emacs-like without requiring it to use Emacs as its frontend. I would like to hit `M-x` in other applications to run commands. I would like to see authors introduce a new version: "Version 3.0, now extendable with Emacsy." I would like hear power users ask, "Yes, but is it Emacsy?"

1.0.2 Motivation

This project was inspired by my frustration creating interactive applications with the conventional edit-run-compile style of development. Finding the right abstraction for the User Interface (UI) that will compose well is not easy. Additionally, If the application is a means to an end and not an end in itself (which is common for academic and in-house tools), then the UI is usually the lowest development priority. Changing the UI is painful, so often mediocre UIs rule. Emacsy allows the developer—or the user—to reshape and extend the UI and application easily at runtime.

1.0.3 Overlooked Treasure

Emacs has a powerful means of programmatically extending itself while it is running. Not many successful applications can boast of that, but I believe a powerful idea within Emacs has been overlooked as an Emacsism rather than an idea of general utility. Let me mention another idea that might have become a Lispism but has since seen widespread adoption.

[1] http://1010.co.uk/gneve.html

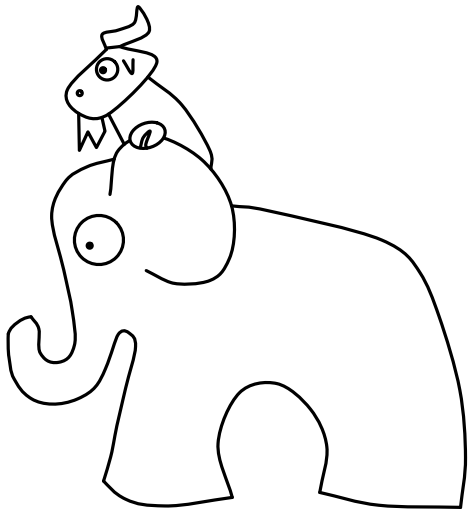


Figure 1.1: The proposed logo features a small gnu riding an elephant.

The Lisp programming language introduced the term Read-Eval-Print-Loop (REPL, pronounced rep-pel), an interactive programming feature present in many dynamic languages: Python, Ruby, MATLAB, Mathematica, Lua to name a few. The pseudo code is given below.

4a ⟨*Lisp REPL* 4a⟩≡

```
(while #t
 (print (eval (read))))
```

The REPL interaction pattern is to enter one complete expression, hit the return key, and the result of that expression will be displayed. It might look like this:

```
> (+ 1 2)
3
```

The kernel of Emacs is conceptually similar to the REPL, but the level of interaction is more fine grained. A REPL assumes a command line interface. Emacs assumes a keyboard interface. I have not seen the kernel of Emacs exhibited in any other applications, but I think it is of similar utility to the REPL—and entirely separate from text editing. I'd like to name this the Key-Lookup-Execute-Command-Loop (KLECL, pronounced clec-cull).

4b ⟨*KLECL* 4b⟩≡

```
(while #t
 (execute-command (lookup-key (read-key))))
```

Long-time Emacs users will be familiar with this idea, but new Emacs users may not be. For instance, when a user hits the 'a' key, then an 'a' is inserted into their document. Let's pull apart the functions to see what that actually looks like with respect to the KLECL.

```
> (read-key)
#\a
> (lookup-key #\a)
self-insert-command
> (execute-command 'self-insert-command)
#t
```

Key sequences in Emacs are associated with commands. The fact that each command is implemented in Lisp is an implementation detail and not essential to the idea of a KLECL.

Note how flexible the KLECL is: One can build a REPL out of a KLECL, or a text editor, or a robot simulator (as shown in the video). Emacs uses the KLECL to create an extensible text editor. Emacsy uses the KLECL to make other applications similarly extensible.

1.0.4 Goals

The goals of this project are as follows.

1. Easy to embed technically

   Emacsy will use Guile Scheme to make it easy to embed within C and C++ programs.

2. Easy to learn

   Emacsy should be easy enough to learn that the uninitiated may easily make parametric changes, e.g., key 'a' now does what key 'b' does and *vice versa*. Programmers in any language ought to be able to make new commands for themselves. And old Emacs hands should be able to happily rely on old idioms and function names to change most anything.

3. Opinionated but not unpersuadable

   Emacsy should be configured with a sensible set of defaults (opinions). Out of the box, it is not *tabla rasa*, a blank slate, where the user must choose every detail, every time. However, if the user wants to choose every detail, they can.

4. Key bindings can be modified

   It wouldn't be Emacs-like if you couldn't tinker with it.

5. Commands can be defined in Emacsy's language or the host language

   New commands can be defined in Guile Scheme or C/C++.

6. Commands compose well

   That is to say, commands can call other commands. No special arrangements must be considered in the general case.

7. A small number of *interface* functions

   The core functions that must be called by the embedding application will be few and straightforward to use.

8. Bring KLECL to light

1.0.5 Anti-goals

Just as important as a project's goals are its anti-goals: the things it is not intended to do.

1. Not a general purpose text editor

   Emacsy will not do general purpose text editing out of the box, although it will have a minibuffer.

2. Not an Emacs replacement

   Emacs is full featured programmer's text editor with more bells and whistles than most people will ever have the time to fully explore. Emacsy extracts the Emacs spirit of application and UI extensibility to use within other programs.

3. Not an Elisp replacement

   There have been many attempts to replace Emacs and elisp with an newer Lisp dialect. Emacsy is not one of them.

4. Not source code compatible with Emacs

   Although Emacsy may adopt some of naming conventions of Emacs, it will not use elisp and will not attempt to be in any way source code compatible with Emacs.

5. Not a framework

   I will not steal your runloop. You call Emacsy when it suits your application not the other way around.

1.1 Emacsy Features

These are the core features from Emacs that will be implemented in Emacsy.

1. keymaps
2. minibuffer
3. recordable macros
4. history
5. tab completion
6. major and minor modes

Chapter 2

The Garden

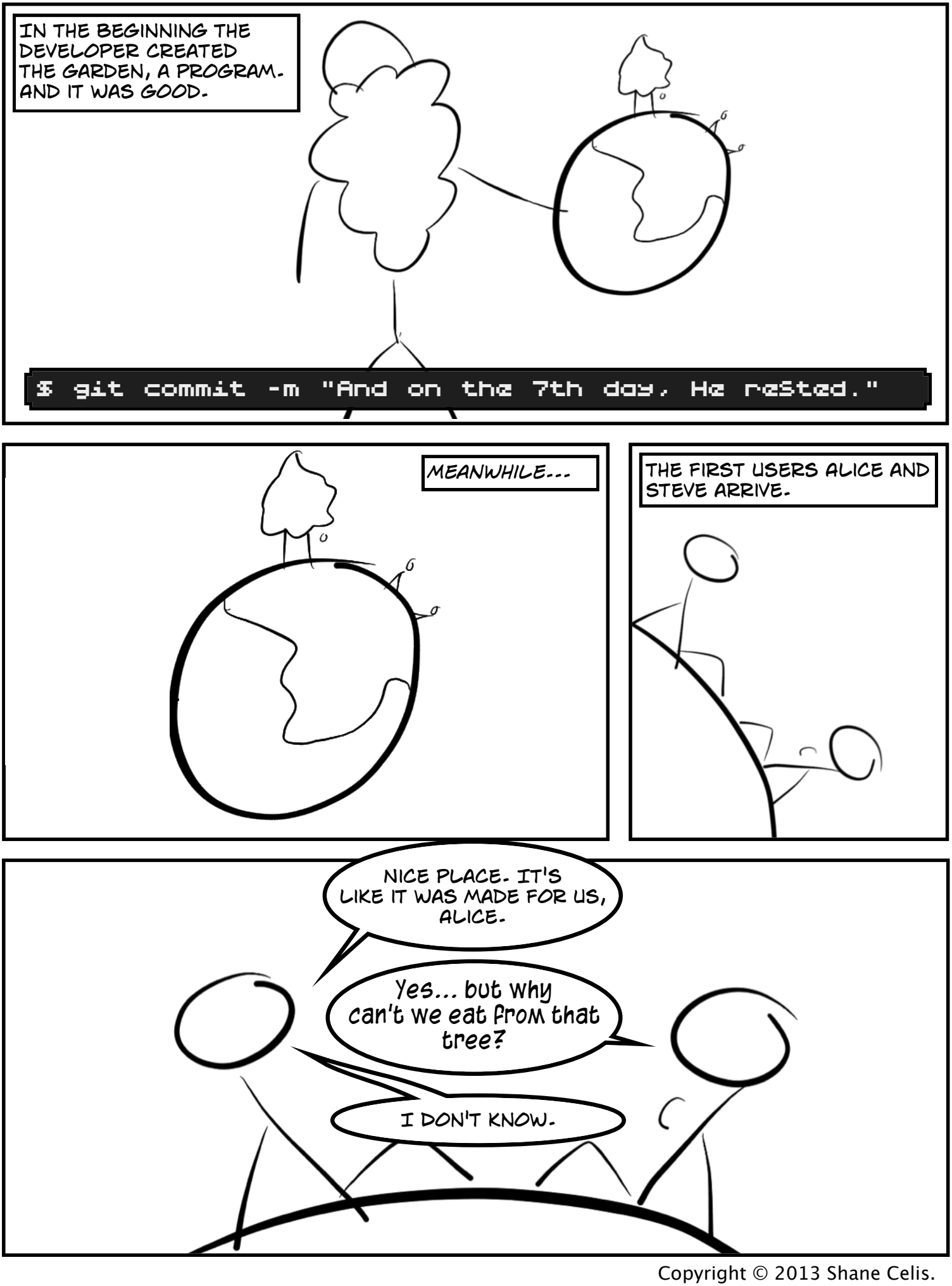
Now for a little entertainment.

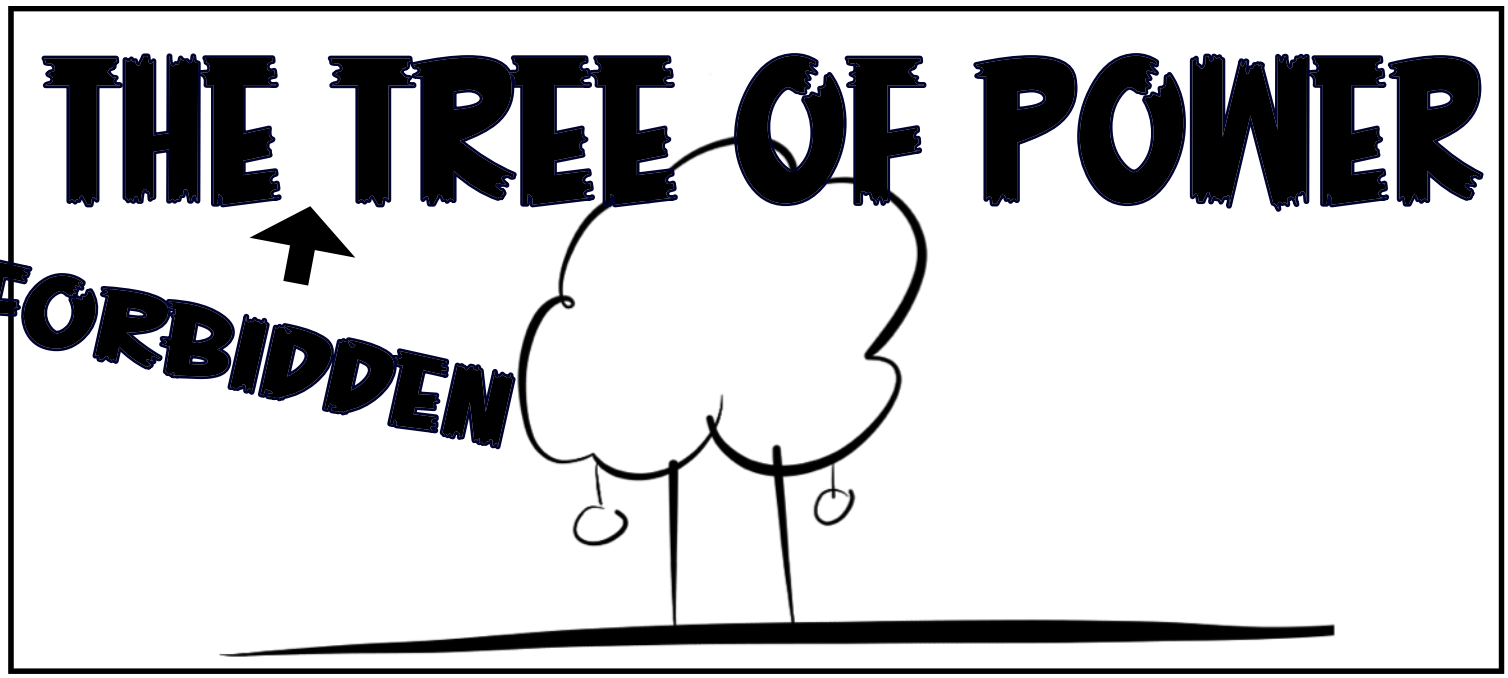
THE GARDEN v 2.0

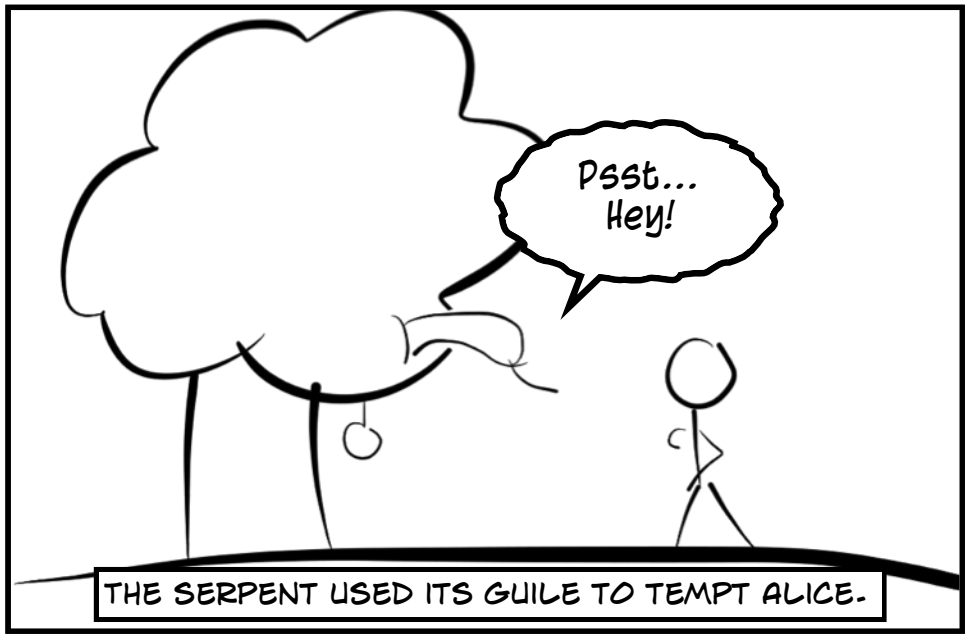
A PARABLE INSPIRED BY

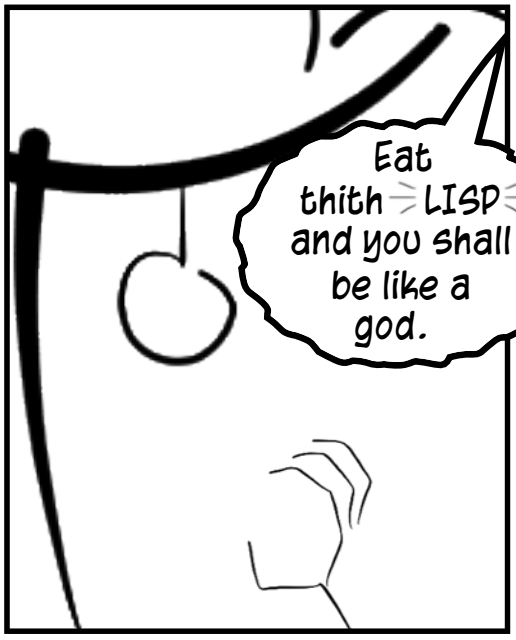
THE CHURCH OF EMACS

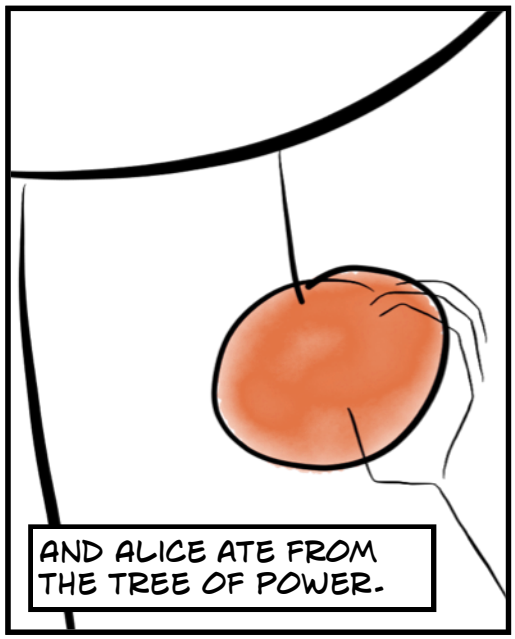
SHANE CELIS

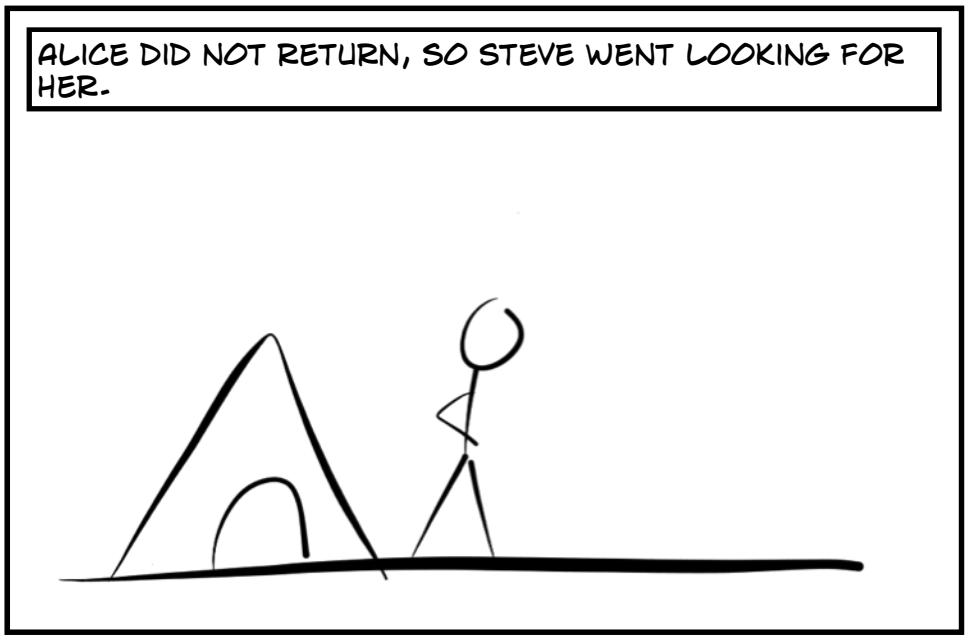


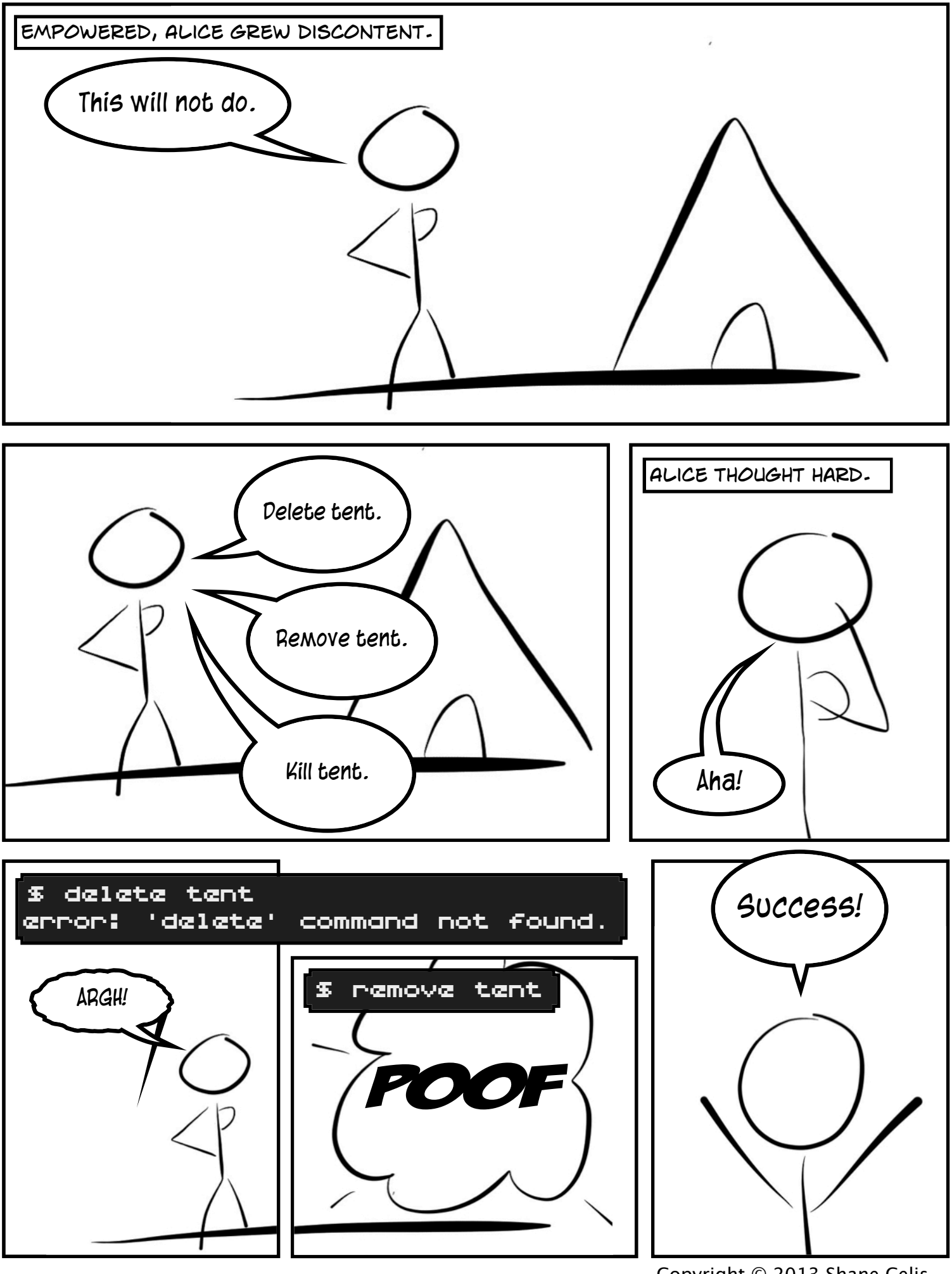




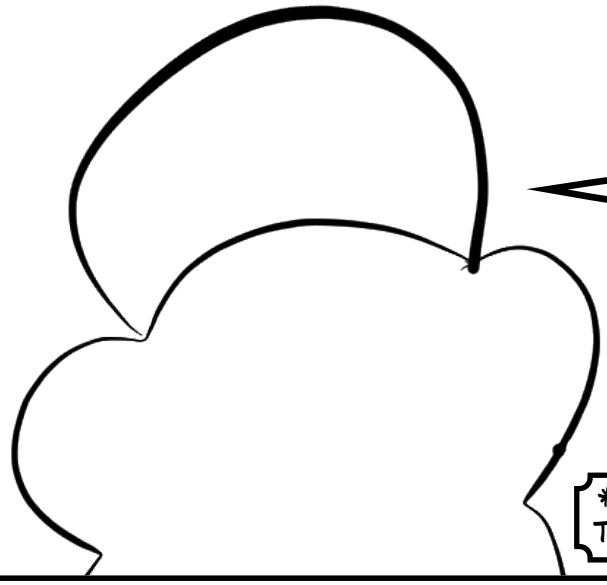






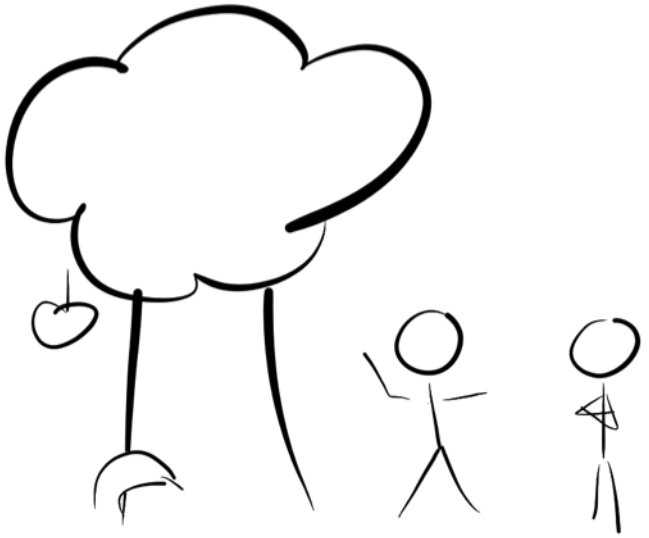


THE DEVELOPER SAW WHAT THEY HAD CREATED. AND IT WAS NOT GOOD.



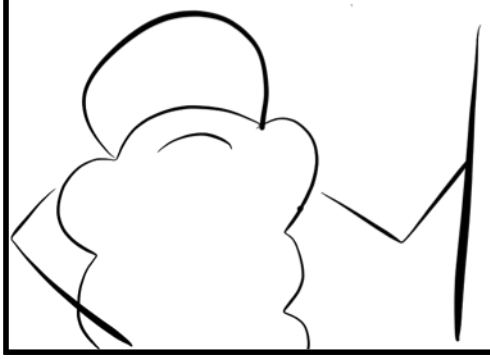
WHAT NEED HAVE YOU TO EMAIL IN MY GARDEN?*

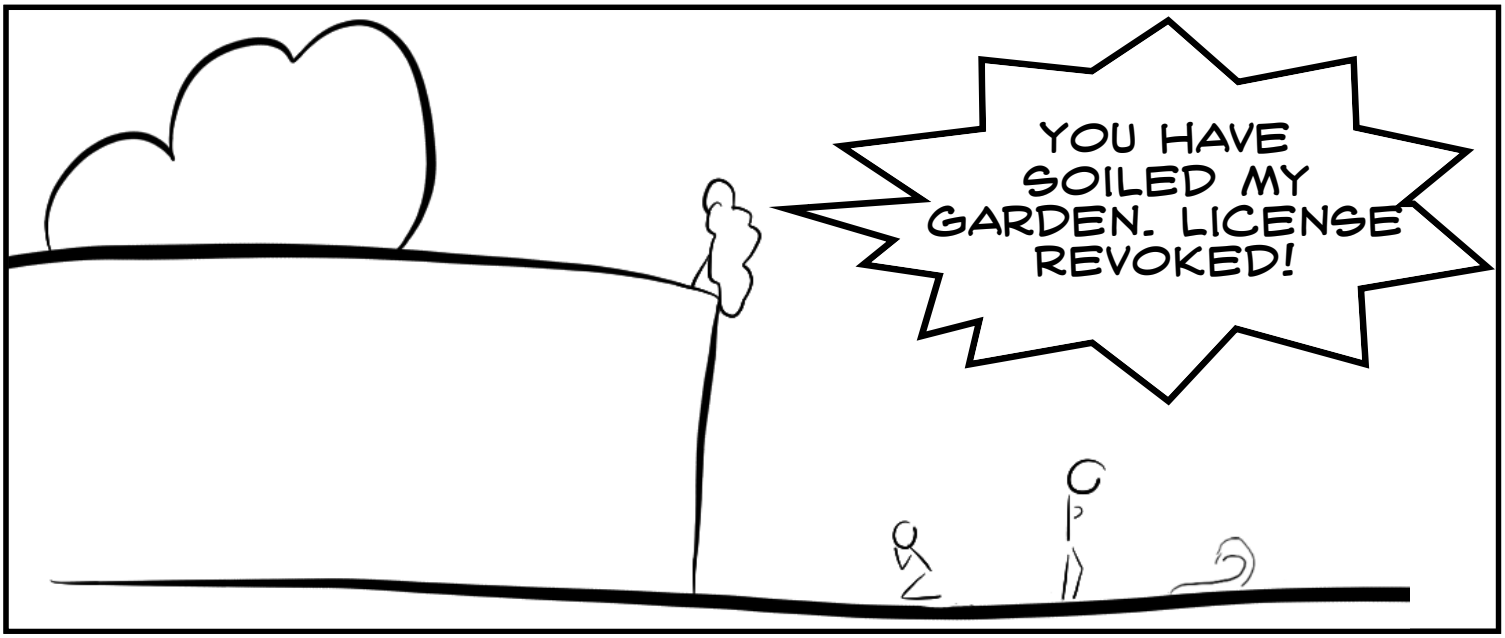
*ZAWINSKI'S LAW: EVERY PROGRAM ATTEMPTS TO EXPAND UNTIL IT CAN READ EMAIL.



STEVE EXPLAINED AND BLAMED.

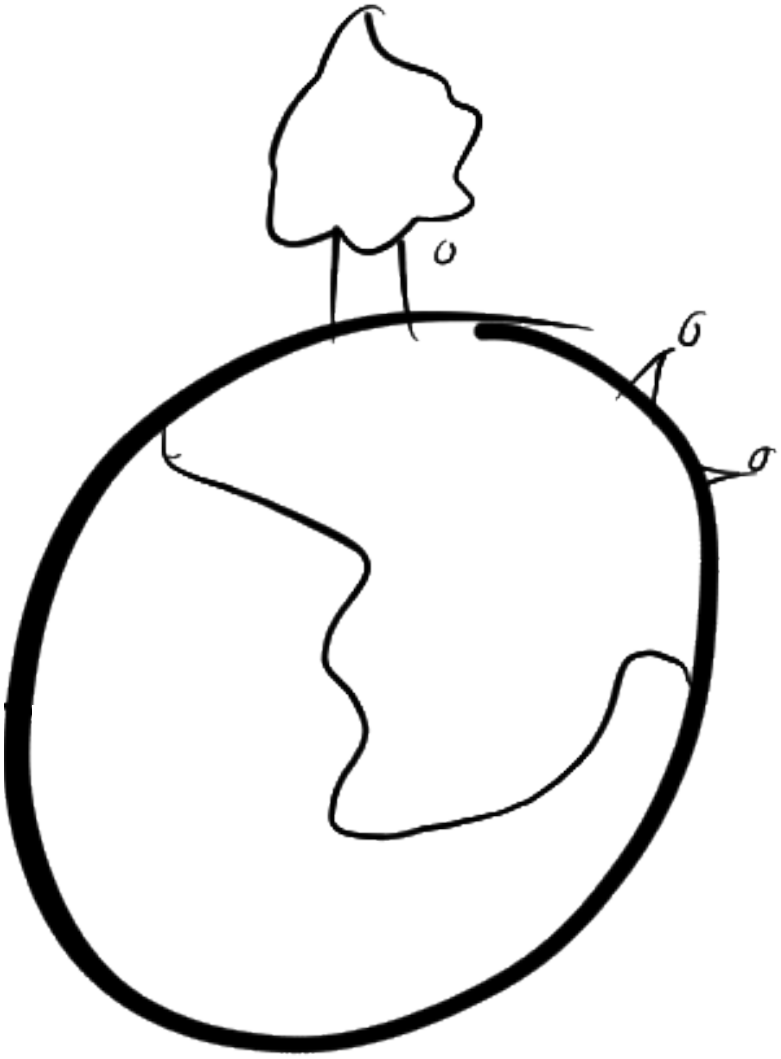
THE DEVELOPER LISTENED AND BECAME ANGRY.





THE END

Parables reveal the truth to some, but hide it from others.
—Dwight Pentecost



SHANE CELIS

Chapter 3

Hello, Emacsy!

3.1 Introduction

I have received a lot of questions asking, what does Emacsy[1] actually do? What restrictions does it impose on the GUI toolkit? How is it possible to not use any Emacs code? I thought it might be best if I were to provide a minimal example program, so that people can see code that illustrates Emacsy API usage.

3.2 Embedders' API

Here are a few function prototypes defined in `emacsy.h`.

[1] Kickstarter page http://kck.st/IY0Bau

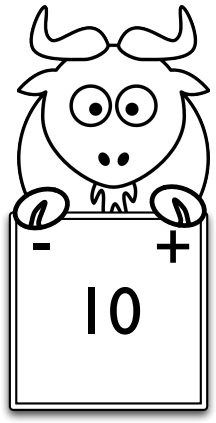


Figure 3.1: Emacsy integrated into the simplest application ever!

18a ⟨*emacsy.h* 18a⟩≡

```
/* Initialize Emacsy. */
int  emacsy_initialize(void);

/* Enqueue a keyboard event. */
void emacsy_key_event(int char_code,
                      int modifier_key_flags);

/* Run an iteration of Emacsy's event loop
   (will not block). */
int emacsy_tick();

/* Return the message or echo area. */
char *emacsy_message_or_echo_area();

/* Return the mode line. */
char *emacsy_mode_line();

/* Terminate Emacsy, runs termination hook. */
int  emacsy_terminate();
```

3.3 The Simplest Application Ever

Let's exercise these functions in a minimal GLUT program we'll call `hello-emacsy`.[2] This simple program will display an integer, the variable `counter`, that one can increment or decrement. The code will be organized as follows.

18b ⟨*file:hello-emacsy.c* 18b⟩≡

⟨*Headers* 22b⟩
⟨*State* 19a⟩
⟨*Functions* 19c⟩
⟨*Primitives* 20b⟩
⟨*Register primitives.* 21a⟩
⟨*Main* 19b⟩

[2]Note: Emacsy does not rely on GLUT. One could use Qt, Cocoa, or ncurses.

Our application's state is captured by one global variable.

19a ⟨*State* 19a⟩≡ (18b)
```
int counter = 0; /* We display this number. */
```

Let's initialize everything in `main` and enter our runloop.

19b ⟨*Main* 19b⟩≡ (18b)
```
int main(int argc, char *argv[]) {
  int err;
  ⟨Initialize GLUT. 23b⟩
  scm_init_guile();    /* Initialize Guile. */
  /* Initialize Emacsy. */
  err = emacsy_initialize();
  if (err)
    exit(err);
  primitives_init();   /* Register primitives. */
  ⟨Load config. 21c⟩
  glutMainLoop();      /* We never return. */
  return 0;
}
```

3.4 Runloop Interaction

Let's look at how Emacsy interacts with your application's runloop since that's probably the most concerning part of embedding. First, let's pass some input to Emacsy.

19c ⟨*Functions* 19c⟩≡ (18b) 20a▷
```
void keyboard_func(unsigned char glut_key,
                   int x, int y) {
  /* Send the key event to Emacsy
     (not processed yet). */
  int key;
  int mod_flags;
  ⟨Get modifier key flags. 23d⟩
  ⟨Handle control modifier. 19d⟩
  emacsy_key_event(key,
                   mod_flags);
  glutPostRedisplay();
}
```

The keys `C-a` and `C-b` returns 1 and 2 respectively. We want to map these to their actual character values.

19d ⟨*Handle control modifier.* 19d⟩≡ (19c)
```
key = mod_flags & EY_MODKEY_CONTROL
  ? glut_key + ('a' - 1)
  : glut_key;
```

The function `display_func` is run for every frame that's drawn. It's effectively our runloop, even though the actual runloop is in GLUT.

20a ⟨*Functions* 19c⟩+≡ (18b) ◁19c 22c▷

```
/* GLUT display function */
void display_func() {
  ⟨Setup display. 23a⟩
  ⟨Display the counter variable. 23c⟩

  /* Process events in Emacsy. */
  if (emacsy_tick() & EY_QUIT_APPLICATION) {
    emacsy_terminate();
    exit(0);
  }

  /* Display Emacsy message/echo area. */
  draw_string(0, 5, emacsy_message_or_echo_area());
  /* Display Emacsy mode line. */
  draw_string(0, 30, emacsy_mode_line());

  glutSwapBuffers();
}
```

At this point, our application can process key events, accept input on the minibuffer, and use nearly all of the facilities that Emacsy offers, but it can't change any application state, which makes it not very interesting yet.

3.5 Plugging Into Your App

Let's define a new primitive Scheme procedure `get-counter`, so Emacsy can access the application's state. This will define a `C` function `SCM scm_get_counter(void)` and a Scheme procedure `(get-counter)`.

20b ⟨*Primitives* 20b⟩≡ (18b) 20c▷

```
SCM_DEFINE (scm_get_counter, "get-counter",
            /* required arg count    */ 0,
            /* optional arg count    */ 0,
            /* variable length args? */ 0,
            (),
            "Returns value of counter.")
{
  return scm_from_int(counter);
}
```

Let's define another primitive Scheme procedure to alter the application's state.

20c ⟨*Primitives* 20b⟩+≡ (18b) ◁20b

```
SCM_DEFINE (scm_set_counter_x, "set-counter!",
         /* required, optional, var. length? */
            1, 0, 0,
            (SCM value),
            "Sets value of counter.")
{
  counter = scm_to_int(value);
  glutPostRedisplay();
  return SCM_UNSPECIFIED;
}
```

Once we have written these primitive procedures, we need to register them with the Scheme runtime.

21a ⟨*Register primitives.* 21a⟩≡ (18b)
```
void primitives_init()
{
#ifndef SCM_MAGIC_SNARFER
  #include "hello-emacsy.c.x"
#endif
}
```

We generate the file `hello-emacsy.x` by running the command: `guile-snarf hello-emacsy.c`. Emacsy can now access and alter the application's internal state.

3.6 Changing the UI

Now let's use these new procedures to create interactive commands and bind them to keys by changing our config file `.hello-emacsy.scm`.

21b ⟨*file:.hello-emacsy.scm* 21b⟩≡ 21d▷
```
(use-modules (emacsy emacsy))

(define-interactive (incr-counter)
 (set-counter! (1+ (get-counter))))

(define-interactive (decr-counter)
 (set-counter! (1- (get-counter))))

(define-key global-map
 (kbd "=") 'incr-counter)
(define-key global-map
 (kbd "-") 'decr-counter)
```

We load this file in `main` like so.

21c ⟨*Load config.* 21c⟩≡ (19b)
```
if (access(".hello-emacsy.scm", R_OK) != -1) {
  scm_c_primitive_load(".hello-emacsy.scm");
} else {
  fprintf(stderr, "warning: unable to load '.hello-emacsy.scm'.\n");
}
```

We can now hit `-` and `=` to decrement and increment the `counter`. This is fine, but what else can we do with it? We could make a macro that increments 5 times by hitting `C-x ( = = = = = C-x )`, then hit `C-e` to run that macro.

Let's implement another command that will ask the user for a number to set the counter to.

21d ⟨*file:.hello-emacsy.scm* 21b⟩+≡ ◁21b 22a▷
```
(define-interactive (change-counter)
 (set-counter!
   (string->number
     (read-from-minibuffer
       "New counter value: "))))
```

Now we can hit `M-x change-counter` and we'll be prompted for the new value we want. There we have it. We have made the simplest application ever more *Emacs-y*.

3.7 Changing it at Runtime

We can add commands easily by changing and reloading the file. But we can do better. Let's start a REPL we can connect to.

22a ⟨*file:.hello-emacsy.scm* 21b⟩+≡ ◁21d

```
(use-modules (system repl server))

;; Start a server on port 37146.
(spawn-server)
```

Now we can `telnet localhost 37146` to get a REPL.

3.8 Conclusion

We implemented a simple interactive application that displays a number. We embedded Emacsy into it: sending events to Emacsy and displaying the minibuffer. We implemented primitive procedures so Emacsy could access and manipulate the application's state. We extended the user interface to accept new commands `+` and `-` to change the state.

3.A Plaintext Please

Here are the plaintext files: emacsy.h, hello-emacsy.c, emacsy-stub.c, and .hello-emacsy.scm. Or

3.B Uninteresting Code

Not particularly interesting bits of code but necessary to compile.

22b ⟨*Headers* 22b⟩≡ (18b)

```
#ifndef SCM_MAGIC_SNARFER
#ifdef __APPLE__
#include <GLUT/glut.h>
#else
#include <GL/glut.h>
#endif
#include <stdlib.h>
#include <emacsy.h>
#endif
#include <libguile.h>

void draw_string(int, int, char*);
```

22c ⟨*Functions* 19c⟩+≡ (18b) ◁20a

```
/* Draws a string at (x, y) on the screen. */
void draw_string(int x, int y, char *string) {
  glLoadIdentity();
  glTranslatef(x, y, 0.);
  glScalef(0.2, 0.2, 1.0);
  while(*string)
    glutStrokeCharacter(GLUT_STROKE_ROMAN,
                        *string++);
}
```

Setup the display buffer the drawing.

23a ⟨*Setup display.* 23a⟩≡ (20a)
```
glClear(GL_COLOR_BUFFER_BIT);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(0.0, 500.0, 0.0, 500.0, -2.0, 500.0);
gluLookAt(0,   0,   2,
          0.0, 0.0, 0.0,
          0.0, 1.0, 0.0);

glMatrixMode(GL_MODELVIEW);
glColor3f(1, 1, 1);
```

23b ⟨*Initialize GLUT.* 23b⟩≡ (19b)
```
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_RGB|GLUT_DOUBLE);
glutInitWindowSize(500, 500);
glutCreateWindow("Hello, Emacsy!");
glutDisplayFunc(display_func);
glutKeyboardFunc(keyboard_func);
```

Our application has just one job.

23c ⟨*Display the counter variable.* 23c⟩≡ (20a)
```
char counter_string[255];
sprintf(counter_string, "%d", counter);
draw_string(250, 250, counter_string);
```

23d ⟨*Get modifier key flags.* 23d⟩≡ (19c)
```
int glut_mod_flags = glutGetModifiers();
mod_flags = 0;
if (glut_mod_flags & GLUT_ACTIVE_SHIFT)
   mod_flags |= EY_MODKEY_SHIFT;
if (glut_mod_flags & GLUT_ACTIVE_CTRL)
   mod_flags |= EY_MODKEY_CONTROL;
if (glut_mod_flags & GLUT_ACTIVE_ALT)
   mod_flags |= EY_MODKEY_META;
```

Chapter 4

C API

Emacsy provides a C API to ease integration with C and C++ programs. The C API is given below.

26 ⟨*Prototypes* 26⟩≡

```
/* Initialize Emacsy. */
int  emacsy_initialize(int init_flags);

/* Enqueue a keyboard event. */
void emacsy_key_event(int char_code,
                      int modifier_key_flags);

/* Enqueue a mouse event. */
void emacsy_mouse_event(int x, int y,
                        int state,
                        int button,
                        int modifier_key_flags);

/* Run an iteration of Emacsy's event loop
   (will not block). */
int emacsy_tick();

/* Return the message or echo area. */
char *emacsy_message_or_echo_area();

/* Return the mode line. */
char *emacsy_mode_line();

/* Return the name of the current buffer. */
char *emacsy_current_buffer();

/* Run a hook. */
int  emacsy_run_hook_0(const char *hook_name);

/* Return the minibuffer point. */
int  emacsy_minibuffer_point();

/* Terminate Emacsy, runs termination hook. */
int  emacsy_terminate();

/* Attempt to load a module. */
int emacsy_load_module(const char *module_name);

/* Load a file in the emacsy environment. */
//int emacsy_load(const char *file_name);

/* Convert the modifier_key_flags into a Scheme list of symbols. */
// Do I want to include any Scheme objects or keep it strictly C?
#include <libguile.h>
SCM modifier_key_flags_to_list(int modifier_key_flags);
```

Uses `emacsy_message_or_echo_area` 30b 30c.

27a ⟨*file:emacsy.h* 27a⟩≡

```
/* emacsy.h

⟨+ Copyright (never defined)⟩

⟨+ License (never defined)⟩
*/

⟨Begin Header Guard 27d⟩

⟨Defines 27b⟩

⟨Prototypes 26⟩

⟨End Header Guard 27e⟩
```

Here are the constants for the C API.

27b ⟨*Defines* 27b⟩≡

```
#define EMACSY_MODKEY_COUNT   6

#define EMACSY_MODKEY_ALT     1 // A
#define EMACSY_MODKEY_CONTROL 2 // C
#define EMACSY_MODKEY_HYPER   4 // H
#define EMACSY_MODKEY_META    8 // M
#define EMACSY_MODKEY_SUPER  16 // s
#define EMACSY_MODKEY_SHIFT  32 // S

#define EMACSY_MOUSE_BUTTON_DOWN  0
#define EMACSY_MOUSE_BUTTON_UP    1
#define EMACSY_MOUSE_MOTION       2

#define EMACSY_INTERACTIVE        1
#define EMACSY_NON_INTERACTIVE    0
```

Here are the return flags that may be returned by `emacsy_tick`.

27c ⟨*Defines* 27b⟩+≡

```
#define EMACSY_QUIT_APPLICATION_P       1
#define EMACSY_ECHO_AREA_UPDATED_P      2
#define EMACSY_MODELINE_UPDATED_P       4
#define EMACSY_RAN_UNDEFINED_COMMAND_P  8
```

The boilerplate guards so that a C++ program may include `emacsy.h` are given below.

27d ⟨*Begin Header Guard* 27d⟩≡

```
#ifdef __cplusplus
 extern "C" {
#endif
```

27e ⟨*End Header Guard* 27e⟩≡

```
#ifdef __cplusplus
 }
#endif
```

The implementation of the API calls similarly named Scheme procedures.

4.1 emacsy_initialize

28a ⟨*Functions* 28a⟩≡
```
int emacsy_initialize(int init_flags)
{
  /* Load the (emacsy emacsy) module. */
  const char *module = "emacsy emacsy";
  int err = emacsy_load_module(module);
  if (err)
    return err;

  (void) scm_call_1(scm_c_public_ref("emacsy emacsy", "emacsy-initialize"),
                    (init_flags & EMACSY_INTERACTIVE) ? SCM_BOOL_T : SCM_BOOL_F);

  return err;
}
```

The function `scm_c_use_module` throws an exception if it cannot find the module, so we have to split that functionality into a body function `load_module_try` and an error handler `load_module_error`.

28b ⟨*Utility Functions* 28b⟩≡
```
SCM load_module_try(void *data)
{
  scm_c_use_module((const char *)data);
  return scm_list_1(SCM_BOOL_T);
}
```

28c ⟨*Utility Functions* 28b⟩+≡
```
SCM load_module_error(void *data, SCM key, SCM args)
{
  //fprintf(stderr, "error: Unable to load module (%s).\n", (const char*) data);
  return scm_list_3(SCM_BOOL_F, key, args);
}
```

Attempt to load a module. Returns 0 if no errors, and non-zero otherwise.

28d ⟨*Utility Functions* 28b⟩+≡
```
int emacsy_load_module(const char *module)
{
  SCM result = scm_internal_catch(SCM_BOOL_T,
                                  load_module_try,   (void *) module,
                                  load_module_error, (void *) module);
  if (scm_is_false(scm_car(result))) {
    fprintf(stderr, "error: Unable to load module (%s); got error to key %s with args %s. Try setting
                    "GUILE_LOAD_PATH environment variable.\n", module,
                    scm_to_locale_string(scm_car(scm_cdr(result))),
                    scm_to_locale_string(scm_car(scm_cdr(scm_cdr(result))))
                    );
    return 1; //EMACSY_ERR_NO_MODULE;
  }
  return 0;
}
```

4.2 emacsy_key_event

29a ⟨*Functions* 28a⟩+≡

```
void emacsy_key_event(int char_code,
                      int modifier_key_flags)
{
  SCM i = scm_from_int(char_code);
  //fprintf(stderr, "i = %d\n", scm_to_int(i));
  SCM c = scm_integer_to_char(i);
  //fprintf(stderr, "c = %d\n", scm_to_int(scm_char_to_integer(c)));

  (void) scm_call_2(scm_c_public_ref("emacsy emacsy", "emacsy-key-event"),
                    c,
                    modifier_key_flags_to_list(modifier_key_flags));
}
```

4.3 emacsy_mouse_event

29b ⟨*Functions* 28a⟩+≡

```
void emacsy_mouse_event(int x, int y,
                        int state,
                        int button,
                        int modifier_key_flags)
{

  SCM down_sym   = scm_c_string_to_symbol("down");
  SCM up_sym     = scm_c_string_to_symbol("up");
  SCM motion_sym = scm_c_string_to_symbol("motion");
  SCM state_sym;
  switch(state) {
  case EMACSY_MOUSE_BUTTON_UP:   state_sym = up_sym;     break;
  case EMACSY_MOUSE_BUTTON_DOWN: state_sym = down_sym;   break;
  case EMACSY_MOUSE_MOTION:      state_sym = motion_sym; break;
  default:
    fprintf(stderr, "warning: mouse event state received invalid input %d.\n",
            state);
    return;
  }

  (void) scm_call_3(scm_c_public_ref("emacsy emacsy", "emacsy-mouse-event"),
                    scm_vector(scm_list_2(scm_from_int(x),
                                          scm_from_int(y))),
                    scm_from_int(button),
                    state_sym);
}
```

4.4 emacsy_tick

30a ⟨*Functions* 28a⟩+≡
```
int emacsy_tick()
{
  int flags = 0;
  (void) scm_call_0(scm_c_public_ref("emacsy emacsy",
                                     "emacsy-tick"));
  if (scm_is_true(scm_c_public_ref("emacsy emacsy",
                                   "emacsy-quit-application?")))
    flags |= EMACSY_QUIT_APPLICATION_P;
  if (scm_is_true(scm_c_public_ref("emacsy emacsy",
                                   "emacsy-ran-undefined-command?")))
    flags |= EMACSY_RAN_UNDEFINED_COMMAND_P;

  return flags;
}
```

4.5 emacsy_message_or_echo_area

30b ⟨*Functions* 28a⟩+≡
```
char *emacsy_message_or_echo_area()
{
  return scm_to_locale_string(
    scm_call_0(scm_c_public_ref("emacsy emacsy",
                                "emacsy-message-or-echo-area")));
}
```
Defines:
`emacsy_message_or_echo_area`, used in chunk 26.

4.6 emacsy_current_buffer

30c ⟨*Functions* 28a⟩+≡
```
char *emacsy_current_buffer()
{
  return scm_to_locale_string(
    scm_call_1(scm_c_public_ref("emacsy emacsy", "buffer-name"),
      scm_call_0(scm_c_public_ref("emacsy emacsy",
                                  "current-buffer"))));
}
```
Defines:
`emacsy_message_or_echo_area`, used in chunk 26.

4.7 emacsy_mode_line

Keep name as modeline.

31a ⟨*Functions* 28a⟩+≡
```
char *emacsy_mode_line()
{
  return scm_to_locale_string(
    scm_call_0(scm_c_public_ref("emacsy emacsy",
                                "emacsy-mode-line")));
}
```

4.8 emacsy_terminate

31b ⟨*Functions* 28a⟩+≡
```
int emacsy_terminate()
{
  SCM result;
  result = scm_call_0(scm_c_public_ref("emacsy emacsy",
                                       "emacsy-terminate"));
  return 0;
}
```

31c ⟨*file:emacsy.c* 31c⟩≡
```
#include "emacsy.h"
#include <libguile.h>

⟨Utility Functions 28b⟩

⟨Functions 28a⟩
```

31d ⟨*Functions* 28a⟩+≡
```
int  emacsy_run_hook_0(const char *hook_name)
{
  /* This should be protected from all sorts of errors that the hooks
     could throw. */
  SCM result;
  result = scm_call_1(scm_c_public_ref("emacsy emacsy",
                                       "emacsy-run-hook"),
                      scm_c_private_ref("guile-user",
                                        hook_name));
  return 0;
}
```

31e ⟨*Functions* 28a⟩+≡
```
int  emacsy_minibuffer_point()
{
  return scm_to_int(
    scm_call_0(scm_c_public_ref("emacsy emacsy",
                                "emacsy-minibuffer-point")));
}
```

32 ⟨*Utility Functions* 28b⟩+≡

```
SCM scm_c_string_to_symbol(const char* str) {
  return scm_string_to_symbol(scm_from_locale_string(str));
}

SCM modifier_key_flags_to_list(int modifier_key_flags)
{
  const char* modifiers[] = { "alt", "control", "hyper", "meta", "super", "shift" };
  SCM list = SCM_EOL;
  for (int i = 0; i < EMACSY_MODKEY_COUNT; i++) {
    if (modifier_key_flags & (1 << i)) {
      list = scm_cons(scm_c_string_to_symbol(modifiers[i]), list);
    }
  }

  return list;
}

SCM_DEFINE(scm_modifier_key_flags_to_list, "modifier-key-flags->list",
           1, 0, 0,
           (SCM flags),
           "Convert an integer of modifier key flags to a list of symbols.")
{
  int modifier_key_flags = scm_to_int(flags);
  return modifier_key_flags_to_list(modifier_key_flags);
}
```

Chapter 5

KLECL

I expounded on the virtues of the Key Lookup Execute Command Loop (KLECL) in 1.0.3. Now we're going to implement a KLECL, which requires fleshing out some concepts. We need events, keymaps, and commands. Let's begin with events.

5.1 Event Module

Let's define an root event class.

Rename time to event-time.

34a ⟨*event:class* 34a⟩≡
```
(define-class-public <event> ()
  (time #:getter time #:init-thunk (lambda () (emacsy-time)))
  (discrete-event? #:getter discrete-event? #:init-keyword #:discrete-event? #:init-value #t))
(export time discrete-event?)
```

This defines the class `<event>`. It relies on `emacsy-time` that I'm going to place in a utilities module, which will be mostly a bucket of miscellaneous things that any module might end up making use of.

34b ⟨*util:procedure* 34b⟩≡
```
(define-public (emacsy-time)
  (exact->inexact (/ (tms:clock (times)) internal-time-units-per-second)))
```

5.1.1 Key Event

Now let's give ourselves an event that'll capture key strokes including the modifier keys.

34c ⟨*event:class* 34a⟩+≡
```
(define-class-public <modifier-key-event> (<event>)
  (modifier-keys #:getter modifier-keys
                 #:init-keyword #:modifier-keys
                 #:init-value '()))

(define-class-public <key-event> (<modifier-key-event>)
  (command-char #:getter command-char
                #:init-keyword #:command-char))
(export modifier-keys command-char)
```

34d ⟨*event:test* 34d⟩≡
```
(check-true (make <key-event> #:command-char #\a))
```

One of the idioms we want to capture from Emacs is this.

```
(define-key global-map "M-f" 'some-command)
```

They `keymap` and `command` module will deal with most of the above, except for the `kbd` procedure. That's something events will be concerned with. One may define a converter for a `kbd-entry` to an event of the proper type. Note that a `kbd-string` is broken into multiple `kbd-entries` on whitespace boundaries, e.g., "C-x C-f" is a `kbd-string` that when parsed becomes two `kbd-entries` "C-x" and "C-f".

Let's write the converter for the `<key-event>` class that will accept the same kind of strings that Emacs does. If the `kbd-entry` does not match the event-type, we return false `#f`.

35a ⟨*event:procedure* 35a⟩≡

```
(define (kbd-entry->key-event kbd-entry)
  (match (strip-off-modifier-keys kbd-entry)
    ((mod-keys kbd-entry)
     (let ((regex "^([^ ]|RET|DEL|ESC|TAB|SPC)$"))
       (let ((match (string-match regex kbd-entry)))
         (if match
             (let* ((char (string->command-char (match:substring match 1))))
               (make <key-event> #:command-char char #:modifier-keys mod-keys))
             #f))))))

(define (get-modifier-keys str)
      (if str
          (map modifier-char->symbol
               (filter (lambda (x) (not (char=? x #\-))) (string->list str)))
          '()))

(define-public (strip-off-modifier-keys kbd-entry)
  "Parse the kbd-entry and strip off the modifier-keys and return the kbd-entry
and a list of modifier keys."
  (let ((regex "^(([ACHMsS]-)*)(.*)$"))
    (let ((match (string-match regex kbd-entry)))
      (if match
          (let ((mod-keys (get-modifier-keys (match:substring match 1))))
            (list mod-keys (match:substring match 3)))
          (list '() kbd-entry)))))
```

35b ⟨*event:test* 34d⟩+≡

```
(check (strip-off-modifier-keys "C-a") => '((control) "a"))
(check (strip-off-modifier-keys "a") => '(() "a"))
(check (strip-off-modifier-keys "asdf") => '(() "asdf"))
```

For the modifier keys, we are going to emulate Emacs to a fault.

35c ⟨*event:procedure* 35a⟩+≡

```
(define-public (modifier-char->symbol char)
  (case char
    ((#\A) 'alt)
    ((#\C) 'control)
    ((#\H) 'hyper)
    ((#\M) 'meta)
    ((#\s) 'super)
    ((#\S) 'shift)
    (else (warn (format #f "Invalid character for modifier key: ~a" char))
          #f)))
```

36a ⟨*event:test* 34d⟩+≡
```
(check (modifier-char->symbol #\S) => 'shift)
(check (modifier-char->symbol #\X) => #f)
```

36b ⟨*event:procedure* 35a⟩+≡
```
(define (string->command-char str)
  (if (= (string-length str) 1)
      ;; One character string, return first character; simple!
      (string-ref str 0)
      (string-case str
                   ("RET" #\cr)
                   ("DEL" #\del)
                   ("ESC" #\esc)
                   ("TAB" #\tab)
                   ("SPC" #\space)
                   (else (warn (format #f "Invalid command character: ~a" str)) ))))
```

Now we have the function `kbd-entry->key-event`. `kbd` needs to know about this and any other converter function. So let's register it.

36c ⟨*event:state* 36c⟩≡
```
(define kbd-converter-functions '())
```

36d ⟨*event:procedure* 35a⟩+≡
```
(define-public (register-kbd-converter function-name function)
  (set! kbd-converter-functions
        (assq-set! kbd-converter-functions function-name function)))
```

Now we can register it.

36e ⟨*event:process* 36e⟩≡
```
(register-kbd-converter 'kbd-entry->key-event kbd-entry->key-event)
```

Rather than doing this for every given converter, let's just write a macro.

36f ⟨*event:macro* 36f⟩≡
```
(define-syntax-public define-kbd-converter
  (syntax-rules ()
    ((define-kbd-converter (name args ...) expr ...)
     (begin (define* (name args ...)
              expr ...)
            (register-kbd-converter 'name name)))
    ((define-kbd-converter name value)
     (begin (define* name value)
            (register-kbd-converter 'name name)))))
```

36g ⟨*event:test* 34d⟩+≡
```
(check-true (memq 'kbd-entry->key-event (alist-keys kbd-converter-functions)))
```

One issue we have with the above is the following:

36h ⟨*event:test* 34d⟩+≡
```
(check (modifier-keys (kbd-entry->key-event "C-C-C-x")) => '(control control control))
```

Our code doesn't account for duplicate modifier keys. For the keymap, we want a unique identifier of an event. Rather than massaging the conversion while in its string form, it seems reasonable to convert the `kbd-entry` into an event, then make the event canonical, then convert back into a string. `kbd` will look like this:

37a ⟨*event:procedure* 35a⟩+≡
```
(define*-public (kbd key-string #:optional (canonical? #t))
  (if canonical?
      (map event->kbd (map canonize-event! (kbd->events key-string)))
      (map event->kbd (kbd->events key-string))))
```

37b ⟨*event:procedure* 35a⟩+≡
```
(define (kbd-entry->event kbd-entry)
    (or (find-first (lambda (f) (f kbd-entry))
                    (alist-values kbd-converter-functions))
        (throw 'invalid-kbd-entry kbd-entry)))
```

The procedure `find-first` is similar to `find`; however, `(find-first f (x . xs))` returns the first `(f x)` that not false rather than the first `x` for which `(f x)` is true.

37c ⟨*util:procedure* 34b⟩+≡
```
(define-public (find-first f lst)
  "Return the first result f which is not false and #f if no such result is found."
  (if (null? lst)
      #f
      (or (f (car lst))
          (find-first f (cdr lst)))))
```

37d ⟨*util:procedure* 34b⟩+≡
```
(define-public (alist-values alist)
  (map cdr alist))

(define-public (alist-keys alist)
  (map car alist))
```

37e ⟨*event:procedure* 35a⟩+≡
```
(define-public (kbd->events kbd-string)
  (let ((kbd-entries (string-tokenize kbd-string)))
       (map kbd-entry->event kbd-entries)))
```

37f ⟨*event:procedure* 35a⟩+≡
```
(define-method-public (canonize-event! (event <key-event>))
  ⟨Deal with shift key. 38a⟩
  (let ((mod-keys (modifier-keys event)))
    ;; Put them in alphabetical order: ACHMsS.
    (slot-set! event 'modifier-keys
               (intersect-order mod-keys
                                '(alt control hyper meta super shift))))
  event)
```

37g ⟨*util:procedure* 34b⟩+≡
```
(define-public (intersect-order list ordered-list )
  "Returns the intersection of the two lists ordered according to the
second argument."
  (filter (lambda (x) (memq x list))
          ordered-list))
```

38a ⟨*Deal with shift key.* 38a⟩≡
```
(if (memq 'shift (modifier-keys event))
    (if (char-set-contains? char-set:requires-shift-key (command-char event))
        ;; Remove extraneous shift.
        (slot-set! event 'modifier-keys (delq 'shift (modifier-keys event)))
        ;; No shift required, but there is a shift in the kbd-entry.
        (if (char-lower-case? (command-char event))
            (begin
              ;; Change the character to uppercase.
              (slot-set! event 'command-char (char-upcase (command-char event)))
              ;; Get rid of the shift.
              (slot-set! event 'modifier-keys (delq 'shift (modifier-keys event)))))))
```

Let's test our canonization of a properly formed but non-canonical event.

38b ⟨*event:test* 34d⟩+≡
```
(let ((key-event (kbd-entry->event "S-C-C-S-a")))
  (check (modifier-keys key-event) => '(shift control control shift))
  (check (command-char key-event) => #\a)
  (canonize-event! key-event)
  (check (modifier-keys key-event) => '(control))
  (check (command-char key-event) => #\A))
```

38c ⟨*event:test* 34d⟩+≡
```
(check (kbd "S-C-C-S-a") => '("C-A"))
(check (kbd "S-C-C-S-A") => '("C-A"))
```

38d ⟨*event:state* 36c⟩+≡
```
(define char-set:requires-shift-key (char-set-union
                                     char-set:symbol
                                     char-set:upper-case
                                     (char-set-delete char-set:punctuation
                                        ;punctuation = !"#%&'()*,-./:;?@[\\]_{}
                                         #\. #\; #\[ #\] #\, #\' #\\)))
```

Now we convert the `<key-event>` back to a `kbd-entry`.

38e ⟨*event:procedure* 35a⟩+≡
```
(define-method-public (event->kbd (event <key-event>))
  (let ((mods (next-method))
        (cmd-char (command-char->string (command-char event))))
    (format #f "~a~a" mods cmd-char)))

(define-method-public (event->kbd (event <modifier-key-event>))
  (let ((mods (map string (map modifier-symbol->char (modifier-keys event)))))
    (string-join `(,@mods "") "-")))
```

38f ⟨*event:test* 34d⟩+≡
```
(check (event->kbd (make <key-event> #:command-char #\a)) => "a")
```

Instead of using `define-generic` I've written a convenience macro `define-generic-public` that exports the symbol to the current module. This mimics the functionality of `define-public`. In general, any *-public macro will export the symbol or syntax to the current module.

39a ⟨*event:procedure* 35a⟩+≡
```
(define-public (modifier-symbol->char sym)
    (case sym
      ((alt) #\A)
      ((control) #\C)
      ((hyper) #\H)
      ((meta) #\M)
      ((super) #\s)
      ((shift) #\S)
      (else (error "Bad modifier symbol " sym))))
```

39b ⟨*event:procedure* 35a⟩+≡
```
(define (command-char->string c)
    (case c
      ((#\cr #\newline) "RET")
      ((#\del) "DEL")
      ((#\esc) "ESC")
      ((#\tab) "TAB")
      ((#\space) "SPC")
      (else (string c))))
```

39c ⟨*event:test* 34d⟩+≡
```
(check (event->kbd (make <key-event> #:command-char #\a
                         #:modifier-keys '(control))) => "C-a")
```

Now we can display the `<key-event>` in a nice way.

39d ⟨*event:procedure* 35a⟩+≡
```
(define-method (write (obj <key-event>) port)
  (display "#<key-event " port)
  (display (event->kbd obj) port)
  (display ">" port))
```

A few procedures to determine whether what kind of objects is nice.

39e ⟨*event:procedure* 35a⟩+≡
```
(define-public (event? obj)
  (is-a? obj <event>))

(define-public (key-event? obj)
  (is-a? obj <key-event>))
```

5.1.2 Mouse Event

We also want to be able to deal with mouse events captured by the class `<mouse-event>`.

39f ⟨*event:class* 34a⟩+≡
```
(define-class-public <mouse-event> (<event>)
  (modifier-keys #:getter modifier-keys #:init-keyword #:modifier-keys #:init-value '())
  (position #:getter position #:init-keyword #:position)
  (button #:getter button #:init-keyword #:button)
  (state #:getter state #:init-keyword #:state))
(export modifier-keys position button state)
```

Mouse drags require a little bit of extra information, for that we have the `<drag-mouse-event>` class.

40a ⟨*event:class* 34a⟩+≡
```
(define-class-public <drag-mouse-event> (<mouse-event>)
  (rect #:getter rect #:init-keyword #:rect))
```

40b ⟨*event:procedure* 35a⟩+≡
```
(define-method (canonize-event! (event <mouse-event>))
  (let ((mod-keys (modifier-keys event)))
    ;; Put them in alphabetical order: ACHMsS.
    (slot-set! event 'modifier-keys
               (intersect-order mod-keys
                                '(alt control hyper meta super shift)))
    event))
```

The `kbd-entry` for mouse events is similar to key events. The regular expression is `^(([ACHMsS]-)*)((up-down-—drag-)?mouse-([123]))$—`.

40c ⟨*event:procedure* 35a⟩+≡
```
(define-kbd-converter (kbd-entry->mouse-event kbd-entry)
  (let* ((regex "^(([ACHMsS]-)*)((up-|down-|drag-)?mouse-([123]))$")
         (match (string-match regex kbd-entry)))
    (if match
        (let* ((symbol (string->symbol (match:substring match 3)))
               (modifier-keys (get-modifier-keys (match:substring match 1))))
          ;; Warning that symbol is not used; squelch with this noop ref.
          symbol
          ⟨Make and return mouse event. 40d⟩)
        ;; It doesn't specify a mouse event; return false.
        #f)))
```

40d ⟨*Make and return mouse event.* 40d⟩≡
```
(make <mouse-event> #:position #f
      #:button (string->number (match:substring match 5))
      #:state (let ((state-string (match:substring match 4)))
                (if state-string
                    (string->symbol
                     (string-trim-right state-string #\-))
                    'click))
      #:modifier-keys modifier-keys)
```

40e ⟨*event:procedure* 35a⟩+≡
```
(define-method (event->kbd (event <mouse-event>))
  (define (state->list state)
    (case state
      ((up down drag)
       (list (symbol->string state)))
      ((click)
       '())
      (else
       (error "Bad state state for mouse event " state))))
  (let ((mods (map string (map modifier-symbol->char (modifier-keys event))))
        (state-list (state->list (state event))))
    (string-join
     `(,@mods ,@state-list "mouse" ,(number->string (button event)))
     "-")))
```

41a ⟨*event:test* 34d⟩+≡
```
(check (kbd "mouse-1") => '("mouse-1"))
(check (kbd "S-S-mouse-1") => '("S-mouse-1"))
```

Finally, let's add some interrogative procedures that mirror Emacs'.

41b ⟨*event:procedure* 35a⟩+≡
```
(define*-public (mouse-event? obj #:optional (of-state #f))
  (and (is-a? obj <mouse-event>)
       (if of-state
           (eq? of-state (state obj))
           #t)))

(define-public (up-mouse-event? e)
  (mouse-event? e 'up))

(define-public (down-mouse-event? e)
  (mouse-event? e 'down))

(define-public (drag-mouse-event? e)
  (mouse-event? e 'drag))

(define-public (click-mouse-event? e)
  (mouse-event? e 'click))

(define-public (motion-mouse-event? e)
  (mouse-event? e 'motion))
```

5.1.3 Dummy Event

Finally, have a dummy event, which is useful to record when used temporal macros.

This should probably be placed in the `kbd-macro` module.

41c ⟨*event:class* 34a⟩+≡
```
(define-class-public <dummy-event> (<event>))
```

41d ⟨*event:procedure* 35a⟩+≡
```
(define-method (canonize-event! (event <event>))
  event)

(define-method (event->kbd (event <event>))
  #f)
```

File Layout

42a ⟨*file:event.scm* 42a⟩≡

```
(define-module (emacsy event)
  #:use-module (ice-9 q)
  #:use-module (ice-9 regex)
  #:use-module (ice-9 optargs)
;  #:use-module (srfi srfi-1)
  #:use-module (ice-9 match)
  #:use-module (oop goops)
  #:use-module (emacsy util)
)
```

⟨*event:macro* 36f⟩
⟨*event:class* 34a⟩
⟨*event:state* 36c⟩
⟨*event:procedure* 35a⟩
⟨*event:process* 36e⟩

Layout for tests.

42b ⟨*file:event-test.scm* 42b⟩≡

```
(use-modules (emacsy event)
             (oop goops)
             )

(eval-when (compile load eval)
           ;; Some trickery so we can test private procedures.
           (module-use! (current-module) (resolve-module '(emacsy event))))
```

⟨*+ Test Preamble* (never defined)⟩
⟨*event:test* 34d⟩
⟨*+ Test Postscript* (never defined)⟩

5.2 Keymap Module

The keymap stores the mapping between key strokes—or events—and commands. Emacs uses lists for its representation of keymaps. Emacsy instead uses a class that stores entries in a hash table. Another difference for Emacsy is that it does not convert `S-C-a` to a different representation like `[33554433]`; it leaves it as a string that is expected to be turned into a canonical representation "C-A".

Here is an example of the keymap representation in Emacs.

```
> (let ((k (make-sparse-keymap)))
    (define-key k "a"         'self-insert-command)
    (define-key k "<mouse-1>" 'mouse-drag-region)
    (define-key k "C-x C-f"   'find-file-at-point)
    k)

(keymap
 (24 keymap
     (6 . find-file-at-point))
 (mouse-1 . mouse-drag-region)
 (97 . self-insert-command))
```

When I initially implemented Emacsy, I replicated Emacs' keymap representation, but I realized it wasn't necessary. And it seems preferrable to make the representation more transparent to casual inspection. Also, Emacsy isn't directly responsible for the conversion of keyboard events into `key-events`—that's a lower level detail that the embedding application must handle. Here is the same keymap as above but in Emacsy.

```
> (let ((k (make-keymap)))
    (define-key k "a"       'self-insert-command)
    (define-key k "mouse-1" 'mouse-drag-region)
    (define-key k "C-x C-f" 'find-file-at-point)
    k)

#<keymap
  a self-insert-command
  C-x #<keymap
        C-f find-file-at-point>
  mouse-1 mouse-drag-region>
```

There are a few differences in how the keymap is produced, and the representation looks slightly different too. For one thing it's not a list.

Justify decisions that deviate from Emacs' design.

Our keymap class has a hashtable of entries and possibly a parent keymap.

44a ⟨*keymap:class* 44a⟩≡
```
(define-class-public <keymap> ()
  (entries #:getter entries #:init-thunk (lambda () (make-hash-table)))
  (parent #:accessor parent #:init-keyword #:parent #:init-value #f))
```

44b ⟨*keymap:test* 44b⟩≡
```
(check-true (make <keymap>))
```

The core functionality of the keymap is being able to define and look up key bindings.

5.2.1 Lookup Key

The procedure `lookup-key` return a keymap or symbol for a given list of keys. Consider this test keymap

44c ⟨*keymap:test* 44b⟩+≡
```
(define (self-insert-command) #f) ;; make a fake command
(define (mouse-drag-region) #f) ;; make a fake command
(define (find-file-at-point) #f) ;; make a fake command
(define k (make-keymap))
(define-key k "a"       'self-insert-command)
(define-key k "mouse-1" 'mouse-drag-region)
(define-key k "C-x C-f" 'find-file-at-point)
```

`lookup-key` should behave in the following way.

44d ⟨*keymap:test* 44b⟩+≡
```
(define (lookup-key* . args)
  (let ((result (apply lookup-key args)))
    (if (procedure? result)
        (procedure-name result)
        result)))
(check (lookup-key* k '("a")) => 'self-insert-command-trampoline)
(check (lookup-key* k "a") => 'self-insert-command-trampoline)
(check (lookup-key k '("b")) => #f)
(check (lookup-key k "M-x b") => #f)
(check-true (keymap? (lookup-key k '("C-x"))))
(check (lookup-key k "C-x C-f a b" #f) => 2)
```

45a ⟨*keymap:procedure* 45a⟩≡
```
(define*-public (lookup-key keymap keys #:optional (follow-parent? #t))
  (define* (lookup-key* keymap keys #:optional (follow-parent? #t))
    (if (null? keys)
        keymap
        (let ((entry (hash-ref (entries keymap) (car keys))))
          (if entry
              (if (keymap? entry)
                  ;; Recurse into the next keymap.
                  (1+if-number (lookup-key* entry (cdr keys) follow-parent?))
                  ;; Entry exists.
                  (if (null? (cdr keys))
                      ;; Specifies the right number of keys; return
                      ;; entry.
                      entry
                      ;; Entry exists but there are more keys; return a
                      ;; number.
                      1))
              ;; No entry; try the parent.
              (if (and follow-parent? (parent keymap))
                  (lookup-key* (parent keymap) keys follow-parent?)
                  ;; No entry; no parent.
                  #f)))))
  (lookup-key* keymap (if (string? keys)
                          (kbd keys)
                          keys) follow-parent?))
```

We propagate the error using a number using the following procedure.

45b ⟨*keymap:procedure* 45a⟩+≡
```
(define (1+if-number x)
  (if (number? x)
      (1+ x)
      x))
```

Because delivering the errors using booleans and numbers is a little cumbersome (and perhaps should be replaced with exceptions?), sometimes we just want to see if there is something in the keymap.

45c ⟨*keymap:test* 44b⟩+≡
```
(check (lookup-key? k "C-x") => #f)
(check (lookup-key? k "C-x C-f") => #t)
(check (lookup-key? k "a") => #t)
```

45d ⟨*keymap:procedure* 45a⟩+≡
```
(define*-public (lookup-key? keymap keyspec #:optional (keymap-ok? #f))
   (let* ((keys (if (string? keyspec)
                    (kbd keyspec)
                    keyspec))
          (result (lookup-key keymap keys)))
     (if keymap-ok?
         (and (not (boolean? result))
              (not (number? result)))
         (and (not (keymap? result))
              (not (boolean? result))
              (not (number? result))))))
```

5.2.2 Define Key

The procedure `define-key` may return a number indicating an error, or a keymap indicating it worked.

46a ⟨*keymap:test* 44b⟩+≡
```
;(check (define-key k (kbd "C-x C-f C-a C-b") 'nope) => 2)
```

46b ⟨*keymap:procedure* 45a⟩+≡
```
(define (make-trampoline module name)
  "Creates a trampoline out of a symbol in a given module, e.g. (lambda () (name))"
  (let ((var (module-variable module name)))
    (unless var
      (scm-error 'no-such-variable "make-trampoline" "Can't make a trampoline for variable named '~a t
    (let ((proc (lambda () ((variable-ref var)))))
      (set-procedure-property! proc 'name
                               (string->symbol (format #f "~a-trampoline" name)))
      proc)))
```

46c ⟨*keymap:procedure* 45a⟩+≡
```
(define-public (define-key keymap key-list-or-string symbol-or-procedure-or-keymap)
  (let* ((keys (if (string? key-list-or-string)
                   (kbd key-list-or-string)
                   key-list-or-string))
         (entry (lookup-key keymap (list (car keys)) #f))
         (procedure-or-keymap
          (if (symbol? symbol-or-procedure-or-keymap)
              (make-trampoline (current-module) symbol-or-procedure-or-keymap)
              symbol-or-procedure-or-keymap)))
    (cond
     ;; Error
     ((number? entry)
      (error "Terminal key binding already found for ~a keys." entry))
     ;; Keymap available for the first key; recurse!
     ((keymap? entry)
      (define-key entry (cdr keys) procedure-or-keymap))
     (else
      (if (= 1 (length keys))
          ;; This is our last key, just add it to our keymap.
          (begin
            (hash-set! (entries keymap) (car keys) procedure-or-keymap)
            keymap)
          ;; We've got a lot of keys left that need to be hung on some
          ;; keymap.
          (define-key keymap (rcdr keys)
            (define-key (make-keymap) (list (rcar keys)) procedure-or-keymap)))))))
```

I use some procedures to access the last item of a list, which I call the `rcar`, and the tail of the list with respect to its end instead of its head `rcdr`. These aren't efficient and should be replaced later.

46d ⟨*util:procedure* 46d⟩≡
```
(define-public (rcar lst)
  (car (reverse lst)))

(define-public (rcdr lst)
  (reverse (cdr (reverse lst))))
```

Let's define a keymap predicate, which is defined in Emacs as `keymapp` ('p' for predicate). I am adopting Scheme's question mark for predicates which seems more natural.

47a ⟨*keymap:procedure* 45a⟩+≡
```
(define-public (keymap? obj)
  (is-a? obj <keymap>))
```

47b ⟨*keymap:test* 44b⟩+≡
```
(check-true  (keymap? (make <keymap>)))
(check-false (keymap? 1))
```

47c ⟨*keymap:procedure* 45a⟩+≡
```
(define*-public (make-keymap #:optional (parent #f))
  (make <keymap> #:parent parent))
```

47d ⟨*keymap:procedure* 45a⟩+≡
```
(define-method (write (obj <keymap>) port)
  (write-keymap obj port))

(define* (write-keymap obj port #:optional (keymap-print-prefix 0))
  (display  "#<keymap " port)
  (hash-for-each (lambda (key value)
                   (do ((i 1 (1+ i)))
                       ((> i keymap-print-prefix))
                     (display " " port))
                   (display "\n" port)
                   (display key port)
                   (display " " port)
                   (if (keymap? value)
                       (write-keymap value port (+ 2 keymap-print-prefix))
                       (display value port)))
                 (entries obj))
  (if (parent obj)
      (write-keymap (parent obj) port (+ 2 keymap-print-prefix)))
  (display ">" port))
```

47e ⟨*keymap:procedure* 45a⟩+≡
```
(define-public (lookup-key-entry? result)
  (and (not (boolean? result)) (not (number? result))))
```

File Layout

47f ⟨*file:keymap.scm* 47f⟩≡
```
(define-module (emacsy keymap)
  #:use-module (ice-9 regex)
  #:use-module (ice-9 optargs)
  #:use-module (oop goops)
  #:use-module (emacsy util)
  #:use-module (emacsy event))
⟨keymap:macro (never defined)⟩
⟨keymap:class 44a⟩
⟨keymap:state (never defined)⟩
⟨keymap:procedure 45a⟩
⟨keymap:process (never defined)⟩
```

Layout for tests.

48 ⟨*file:keymap-test.scm* 48⟩≡

```
(use-modules (emacsy keymap)
             (emacsy event)
             (oop goops))

(eval-when (compile load eval)
           ;; Some trickery so we can test private procedures.
           (module-use! (current-module) (resolve-module '(emacsy keymap))))
```

⟨+ *Test Preamble* (never defined)⟩
⟨*keymap:test* 44b⟩
⟨+ *Test Postscript* (never defined)⟩

5.3 Command Module

> If words of command are not clear and distinct, if orders are not thoroughly understood, then the general is to blame.
>
> Sun Tzu

The command module is responsible for a couple things. In Emacs one defines commands by using the special form `(interactive)` within the body of the procedure. Consider this simple command.

```
(defun hello-command ()
  (interactive)
  (message "Hello, Emacs!"))
```

Emacsy uses a more Scheme-like means of defining commands as shown below.

```
(define-interactive (hello-command)
  (message "Hello, Emacsy!"))
```

One deviation from Emacs I want to see within Emacsy is to have the commands be more context sensitive. To illustrate the problem when I hit `M-x TAB TAB` it autocompletes all the available commands into a buffer. In my case that buffer contains 4,840 commands. This doesn't seem to hurt command usability, but it does hurt the command discoverability.

I want Emacsy to have command sets that are analogous to keymaps. There will be a global command set `global-cmdset` similar to the global keymap `global-map`. And in the same way that major and minor modes may add keymaps to a particular buffer, so too may they add command maps.

Figure out where to look up any given function/variable using this kind of code (apropos-internal "êmacsy.*"). Refer to ice-9 readline package for an example of its usage.

The class holds the entries, a string completer for tab completion, and potentially a parent command map.

Wouldn't this better be thought of as a command set rather than map. Also, having it as a map means there could be two different implementations of the command; the one referred to by the procedure, and the one referred to in the map. They could be become unsynchronized.

49a ⟨*command:class* 49a⟩≡
```
(define-class-public <command-set> ()
  (commands #:getter commands #:init-form (list))
  (completer #:getter completer #:init-form (make <string-completer>))
  (parent #:accessor parent #:init-keyword #:parent #:init-value #f))
(export commands completer)
```

We have accessors for adding, removing, and testing what's in the set. Note that the parent set is never mutated.

49b ⟨*command:procedure* 49b⟩≡
```
(define-method-public (command-contains? (cmap <command-set>) command-symbol)
  (or (memq command-symbol (commands cmap))
      (and (parent cmap) (command-contains? (parent cmap) command-symbol))))

(define-method-public (command-add! (cmap <command-set>) command-symbol)
  (when (not (command-contains? cmap command-symbol))
      (add-strings! (completer cmap) (list (symbol->string command-symbol)))
      (slot-set! cmap 'commands (cons command-symbol (commands cmap)))))

(define-method-public (command-remove! (cmap <command-set>) command-symbol)
  (when (command-contains? cmap command-symbol)
    (slot-set! cmap 'commands (delq! command-symbol (commands cmap)))
    ;; Must rebuild the completer.
    (let ((c (make <string-completer>)))
      (add-strings! c (map symbol->string (commands cmap)))
      (slot-set! cmap 'completer c))))
```

We define the global command map.

49c ⟨*command:state* 49c⟩≡
```
(define-public global-cmdset (make <command-set>))
```

Perhaps procedure-properties should be used to denote a procedure as a command?

50 ⟨*command:macro* 50⟩≡

```
(define-public (module-command-interface mod)
  (unless (module-variable mod '%module-command-interface)
    (module-define! mod '%module-command-interface
                    (let ((iface (make-module)))
                      (set-module-name! iface (module-name mod))
                      (set-module-version! iface (module-version mod))
                      (set-module-kind! iface 'command)
                      ;(module-use! iface (resolve-interface '(guile)))
                      iface)))
  (module-ref mod '%module-command-interface))



(define-public (module-export-command! m names)
  (let ((public-i (module-command-interface m)))
    ;; Add them to this module.
    (for-each (lambda (name)
                (let* ((internal-name (if (pair? name) (car name) name))
                       (external-name (if (pair? name) (cdr name) name))
                       (var (module-ensure-local-variable! m internal-name)))
                  (module-add! public-i external-name var)))
              names)))

(define-syntax-rule (export-command name ...)
  (eval-when (eval load compile expand)
    (call-with-deferred-observers
     (lambda ()
       (module-export-command! (current-module) '(name ...))))))

(define-syntax-public define-interactive
  (syntax-rules ()
    ((define-interactive (name . args) . body)
     (begin (define-cmd global-cmdset (name . args)
              . body)
            (export-command name)))
    ((define-interactive name value)
     (begin (define-cmd global-cmdset name value)
            (export-command name))
     )))
```

Need to fix: define-cmd doesn't respect documentation strings.

51 ⟨*command:macro* 50⟩+≡

```
(define-syntax-public define-cmd
  (lambda (x)
    (syntax-case x ()
      ((define-cmd (name . args) e0)
       #'(begin
           (define* (name . args)
             (with-fluids ((in-what-command 'name))
               e0))
           (export name)
           (emacsy-kind-set!
            (module-variable (current-module) 'name)
            'command)
           (set-command-properties! name 'name)))
      ((define-cmd (name . args) e0 e1 . body)
       (string? (syntax->datum #'e0))
       ;; Handle the case where there is a documentation string.
       #'(begin
           (define* (name . args)
                    e0
                    (with-fluids ((in-what-command 'name))
                      (let ()
                        e1 . body)))
           (export name)
           (emacsy-kind-set!
            (module-variable (current-module) 'name)
            'command)
           (set-command-properties! name 'name)))
      ((define-cmd (name . args) e0 e1 . body)
       #'(begin
           (define* (name . args)
                    (with-fluids ((in-what-command 'name))
                      (let ()
                        e0 e1 . body)))
           (export name)
           (emacsy-kind-set!
            (module-variable (current-module) 'name)
            'command)
           (set-command-properties! name 'name)))
      ((define-cmd name value)
       #'(begin
           (define name #f)
           (let ((v value))
             (set! name (colambda args
                          (with-fluids ((in-what-command 'name))
                            (apply v args))))
             (export name)
             (emacsy-kind-set!
              (module-variable (current-module) 'name)
              'command)
             (set-command-properties! name 'name))))
```

```
((define-cmd cmap (name . args) . body)
 #'(begin
     (define-cmd (name . args) . body)
     (command-add! cmap 'name)))
((define-cmd cmap name value)
 #'(begin
     (define-cmd name value)
     (command-add! cmap 'name))))))
```

52a ⟨*command:procedure* 49b⟩+≡

```
(define-public (register-interactive name proc)
  (command-add! global-cmdset name)
  (set-command-properties! proc name))
```

52b ⟨*command:procedure* 49b⟩+≡

```
(define-public (command->proc command)
  (cond
   ((thunk? command)
    command)
   (else
    (warn "command->proc not given a command: ~a" command)
    #f)))
```

52c ⟨*command:procedure* 49b⟩+≡

```
(define-public (command-name command)
  (procedure-name command))
```

52d ⟨*command:procedure* 49b⟩+≡

```
(define-public (command? object)
  (thunk? object))
```

5.3.1 Determine Interactivity

We would like to be able to determine within the command procedure's body whether the command has been called interactively, by the user's key press, or by a keyboard macro or another procedure call. The best way I can think to do this is to have a means of answering the following questions: 1) What command am I in? 2) What is the current interactive command?

Determining the current command is not that difficult. That's generally set by the `this-command` variable. However, determining what command I am in is a little troublesome. One can examine the stack and look for the first procedure that has some property associated with commands.

52e ⟨*command:procedure* 49b⟩+≡

```
(define* (set-command-properties! proc #:optional (name #f))
  (let ((cname (or name (procedure-name proc) #f)))
    (set-procedure-property! proc 'command-name
                             (if (eq? cname 'proc)
                                 #f
                                 cname))))
```

52f ⟨*command:state* 49c⟩+≡

```
(define in-what-command (make-fluid #f))
```

53a ⟨*command:macro* 50⟩+≡
```
(define-syntax-public lambda-cmd
  (syntax-rules ()
    ((lambda-cmd args . body)
     (let ((proc (lambda* args
                          (with-fluids ((in-what-command #f))
                            . body))))
       (set-command-properties! proc)
       proc))))
```

53b ⟨*command:test* 53b⟩≡
```
(define test-cmd (lambda-cmd args 1))
(define (test-cmd-2) 2)
(define-cmd (test-cmd-3) 3)
(check (procedure-documentation test-cmd-3) => #f)
(check (test-cmd) => 1)
(check-true (command? test-cmd))
(check-true (command? test-cmd-2))
(check-true (command? test-cmd-3))
(check (assq-ref (procedure-properties test-cmd) 'command-name) => #f)
(check (assq 'command-name (procedure-properties test-cmd-2)) => #f)
(check (command-name test-cmd) => 'proc)
(check (command-name test-cmd-2) => 'test-cmd-2)
(check (command-name test-cmd-3) => 'test-cmd-3)
```

53c ⟨*command:procedure* 49b⟩+≡
```
(define-public (what-command-am-i?)
  (fluid-ref in-what-command))
```

53d ⟨*command:test* 53b⟩+≡
```
(define-cmd (test-who-am-i?)
  "test-who-am-i? documentation"
  (let ((w (what-command-am-i?)))
    1
    w))
(check (command-name test-who-am-i?) => 'test-who-am-i?)
(check (test-who-am-i?) => 'test-who-am-i?)
(check (procedure-documentation test-who-am-i?) => "test-who-am-i? documentation")
```

53e ⟨*command:procedure* 49b⟩+≡
```
(define-public (command-execute command . args)
  (if (command? command)
      (let ((cmd-proc (command->proc command))
            (cmd-name (command-name command)))
        (emacsy-log-info "Running command: ~a" cmd-name)
        (set! last-command this-command)
        (set! this-command cmd-name)
        (apply cmd-proc args))
      (error (emacsy-log-warning "command-execute not given a command: ~a" command))))
```

53f ⟨*util:procedure* 53f⟩≡
```
(define-public (emacsy-log-info format-msg . args)
  (apply format (current-error-port) format-msg args)
  (newline (current-error-port)))
```

54a ⟨*command:state* 49c⟩+≡
```
(define-public this-command #f)
(define-public last-command #f)
```

54b ⟨*command:state* 49c⟩+≡
```
(define-public kill-rogue-coroutine? #f)
(define-public seconds-to-wait-for-yield 2)
```

54c ⟨*command:procedure* 49b⟩+≡
```
(define-public (call-interactively command . args)
  (dynamic-wind
   (lambda () (if kill-rogue-coroutine?
                  (alarm seconds-to-wait-for-yield)))
   (lambda () (with-fluids ((this-interactive-command (command-name command)))
                (apply command-execute command args)))
   (lambda () (if kill-rogue-coroutine?
                  (alarm 0)))))
```

54d ⟨*command:state* 49c⟩+≡
```
(define this-interactive-command (make-fluid))
```

54e ⟨*command:procedure* 49b⟩+≡
```
(define*-public (called-interactively? #:optional (kind 'any))
  (eq? (fluid-ref in-what-command) (fluid-ref this-interactive-command)))
```

54f ⟨*command:test* 53b⟩+≡
```
(define-cmd (foo)
  (if (called-interactively?)
      'interactive
      'non-interactive))
(check (command? 'foo) => #f)
(check (command? foo) => #t)
(check (command-name foo) => 'foo)
(check-true (command->proc foo))

(check-throw (command-execute 'foo) => 'misc-error)
(check (command-execute foo) => 'non-interactive)
(check (call-interactively foo) => 'interactive)
```

File Layout

54g ⟨*file:command.scm* 54g⟩≡
```
(define-module (emacsy command)
  #:use-module (string completion)
  #:use-module (oop goops)
  #:use-module (ice-9 optargs)
  #:use-module (emacsy util)
  #:use-module (emacsy self-doc)
  #:use-module (emacsy coroutine)

  #:export-syntax (export-command))
⟨command:macro 50⟩
⟨command:class 49a⟩
⟨command:state 49c⟩
⟨command:procedure 49b⟩
⟨command:process (never defined)⟩
```

Layout for tests.

55 ⟨*file:command-test.scm* 55⟩≡

```
(use-modules (emacsy command)
             (emacsy event)
             (oop goops))

(eval-when (compile load eval)
           ;; Some trickery so we can test private procedures.
           (module-use! (current-module) (resolve-module '(emacsy command))))
```

⟨+ *Test Preamble* (never defined)⟩
⟨*command:test* 53b⟩
⟨+ *Test Postscript* (never defined)⟩

5.4 Block Module

> Wearied I fell asleep: But now lead on; In me is no delay; with thee to go, Is to stay here
>
> Paradise Lost
> John Milton

The `block` module handles blocking in Emacsy. When I prototyped Emacsy, I considered this the riskiest part of the project. If I couldn't get this to work, it wouldn't be worth trying to develop the idea further. To understand what I mean, one can try running the following in Emacs `M-: (read-key)`. This will evaluate `read-key` and effectively block until there is another key press.

Implementing "blocking" on a small set of bare functions can be done without too much trickery. However, what if you have computations that follow after these functions? For instance if you evaluate `M-: (message "Got %s" (read-key))`, `read-key` must block until a key is pressed, then resume the computation that will call `message`. An Operating System must perform a similar operation whenever a system call is made, usually implemented using interrupts or traps. Without recourse to interrupts and bare stack manipulation, what can we do to achieve a similar feature?

GNU Guile has a terrific feature called delimited continuations. Here is an example of a delimited continuation from the Guile Manual. This continuation `cont`

```
(define cont
  (call-with-prompt
   ;; tag
   'foo
   ;; thunk
   (lambda ()
     (+ 34 (abort-to-prompt 'foo)))
   ;; handler
   (lambda (k) k)))
```

could be rewritten as

```
(define cont
  (lambda (x)
    (+ 34 x)))
```

I had to read and re-read this example to let it sink in. What does it buy us? It allows us to abort a computation at any time and resume it later.[1] So if we were to implement `read-key`, we abort the computation if there has been no key press. Our main loop in `C` continues to run, redraw, wait for key presses. When a key press comes, we can resume that computation—that continuation. That's the idea. What's beautiful about this is that the user code has access to the same rich input services as the system code without any unnatural contortions. These "system calls" look like regular procedure calls much like the Unix call to `open` looks like a regular function call.

One of the key features I figured one bought by embedding a higher-level language like Scheme was garbage collection. High-level blocking while still being low-level non-blocking is a huge boon. What we'll implement is a simple blocking system using Guile's delimited continuations, also called prompts.

Let's start with the tests, so the usage is somewhat obvious.

[1] Lua's coroutines also seem like a good candidate for pulling off a trick like this. Python's generators, however, do not.

57a ⟨*block:test* 57a⟩≡
```
(define done-blocking? #f)
(define (i-block)
  (block-yield)
  (set! done-blocking? #t))
```

`i-block` will immediately yield. If it is not called with `call-blockable` then it will throw an error.

57b ⟨*block:test* 57a⟩+≡
```
(check-throw (i-block) => 'misc-error)
```

57c ⟨*block:procedure* 57c⟩≡
```
(define-public (block-yield)
  ;; I forgot why I'm running this thunk.
  (run-thunk (abort-to-prompt 'block 'block-until
                              (const #t) #t)))
```

`call-blockable` will handle any aborts to the `'block` prompt. If the thunk aborts, it adds an instance of the class `<blocking-continuation>` to a list of such instances.

57d ⟨*block:state* 57d⟩≡
```
(define blocking-continuations '())
```

57e ⟨*block:procedure* 57c⟩+≡
```
(define-public (call-blockable thunk)
  (let ((bc #f))
    (call-with-prompt
     'block
     thunk
     (lambda (cc kind . args)
       (case kind
         ((block-until)
          (let ((continue-command-loop? #t)
                (continue-wait? #t))
            (set! bc ⟨Make blocking continuation. 58a⟩)
            ;; Remember this bc.
            (cons! bc blocking-continuations))))))
    bc))
```

57f ⟨*util:state* 57f⟩≡
```
;; I want to get rid of this state if I can.
(define-public continue-command-loop? (make-unbound-fluid))
```

Let's add a little syntactic sugar `with-blockable`.

57g ⟨*block:macro* 57g⟩≡
```
(define-syntax-public with-blockable
  (syntax-rules ()
    ((with-blockable e ...)
     (call-blockable (lambda () e ...)))))
```

58a ⟨*Make blocking continuation.* 58a⟩≡
```
(make <blocking-continuation>
  #:tag 'block-until
  #:continuation cc
  #:loop-number 0
  #:continue-when? (car args)
  #:continue-now
  (lambda ()
    (set! continue-command-loop? #f)
    (if continue-wait?
        (call-blockable
         (lambda () (cc (lambda () #t))))))
  #:serial? (cadr args))
```

Now we can call `i-block` and capture its continuation.

58b ⟨*block:test* 57a⟩+≡
```
(check-true (call-blockable (lambda () (i-block))))
(check (length blocking-continuations) => 1)
```

To possibly resume these continuations, we're going to call `block-tick`. Additionally, continuations come in two flavors: serial and non-serial. The constraints on resuming are different. A non-serial block can be resumed whenever the `continue-when?`thunk return true. A serial block, however, will only be resumed after every other serial block that has a greater number, meaning more recent, has been resumed.

rename continue-now?

58c ⟨*block:procedure* 57c⟩+≡
```
(define-public (block-tick)
  (set! blocking-continuations
        ;; Sort the continuations by the most recent ones.
        (sort! blocking-continuations (lambda (a b)
                                        (> (number a) (number b)))))
    (let ((ran-serial? #f))
      (for-each
       (lambda (bc)
         (if (not (serial? bc))
             ;; If it's not serial, we might run it.
             (maybe-continue bc)
             ;; If it's serial, we only run the top one.
             (if (and (not ran-serial?) (serial? bc))
                 (begin
                   (if (maybe-continue bc)
                       (set! ran-serial? #t))))))
       blocking-continuations))
    ;; Keep everything that hasn't been run.
    (set! blocking-continuations
          (filter! (lambda (bc) (not (ran? bc)))
                   blocking-continuations))
    ;(format #t "blocking-continuations #~a of ~a~%" (length blocking-continuations) (map number bloc
    (when (or (null? blocking-continuations)
              (null? (filter serial? blocking-continuations)))
      (run-hook no-blocking-continuations-hook))
    #t)
```

Maybe get rid of no-blocking-continuations-hook and just have a predicate to test for whether any blocks exist?

59a ⟨*block:procedure* 57c⟩+≡
```
(define*-public (blocking?)
  (> (length blocking-continuations) 0))
```

59b ⟨*block:procedure* 57c⟩+≡
```
(define-method (maybe-continue (obj <blocking-continuation>))
  (if (and (not (ran? obj))
;           (or run-serial? (serial? obj))
           ;; this line crashed.
           (run-thunk (slot-ref obj 'continue-when?)))
      (begin (set! (ran? obj) #t)
             (run-thunk (slot-ref obj 'continue-now))
             #t)
      #f))
```

If there are no blocking continuations, we run this hook.

59c ⟨*block:state* 57d⟩+≡
```
(define-public no-blocking-continuations-hook (make-hook))
```

Now we should be able to resume `i-block` by running `block-tick`.

59d ⟨*block:test* 57a⟩+≡
```
(check done-blocking? => #f)
(check (block-tick) => #t)
(check done-blocking? => #t)
(check (length blocking-continuations) => 0)
```

In addition to simply yielding we can block until a particular condition is met.

59e ⟨*block:procedure* 57c⟩+≡
```
(define*-public (block-until condition-thunk #:optional (serial? #f))
  (if (not (run-thunk condition-thunk))
      (run-thunk (abort-to-prompt 'block 'block-until
                                  condition-thunk serial?))))
```

And if we have `block-until`, it's easy to write `block-while`.

59f ⟨*block:procedure* 57c⟩+≡
```
(define*-public (block-while condition-thunk #:optional (serial? #f))
  (block-until (negate condition-thunk) serial?))
```

Let's exercise this `block-until` procedure.

59g ⟨*block:test* 57a⟩+≡
```
(define continue-blocking? #t)
(define (i-block-until)
  (block-until (lambda () (not continue-blocking?))))
(check (length blocking-continuations) => 0)
(call-blockable (lambda () (i-block-until)))
(check (length blocking-continuations) => 1)
```

Now, even if we call `block-tick` it shouldn't be resumed.

59h ⟨*block:test* 57a⟩+≡
```
(block-tick)
(check (length blocking-continuations) => 1)
```

Let's change the condition for our blocking call.

60a ⟨*block:test* 57a⟩+≡
```
(set! continue-blocking? #f)
(check (length blocking-continuations) => 1)
(block-tick)
(check (length blocking-continuations) => 0)
```

Sometimes we may just want to kill a blocking continuation. One could just forget the reference and let it be garbage collected. Here, we're going to throw an exception such that whatever the continuation was doing can potentially be cleaned up.

60b ⟨*block:procedure* 57c⟩+≡
```
(define-method-public (block-kill (obj <blocking-continuation>))
  (set! (ran? obj) #t)
  (call-blockable
   (lambda () ((slot-ref obj 'continuation)
               (lambda ()
                 (throw 'block-killed obj)
                 #f)))))
```

Let's exercise `block-kill`.

60c ⟨*block:test* 57a⟩+≡
```
(set! continue-blocking? #t)
(let ((bc (call-blockable (lambda () (i-block-until)))))
  (check (length blocking-continuations) => 1)
  (block-tick)
  (check (length blocking-continuations) => 1)
  (check-throw (block-kill bc) => 'block-killed)
  ;; The killed block is not cleaned out immediately.
  (check (length blocking-continuations) => 1)
  (block-tick)
  (check (length blocking-continuations) => 0))
```

We're going to capture these blocking continuations into a class.

60d ⟨*block:class* 60d⟩≡
```
(define-class <blocking-continuation> ()
  (number #:getter number #:init-thunk (let ((count -1))
                                         (lambda () (incr! count))))
  (loop-number #:getter loop-number #:init-keyword #:loop-number)
  (tag #:getter tag #:init-keyword #:tag)
  (continuation #:init-keyword #:continuation)
  (continue-when? #:init-keyword #:continue-when?)
  (continue-now #:init-keyword #:continue-now)
  ;; Has this ran and ready to be deleted?
  (ran? #:accessor ran? #:init-value #f)
  (serial? #:getter serial? #:init-keyword #:serial? #:init-value #t))

(define-method (write (obj <blocking-continuation>) port)
  (write (string-concatenate
          (list "#<bc " (symbol->string (tag obj))
                " " (number->string (number obj))
                " cl " (number->string (loop-number obj)) ">")) port))
```

There's a lot of information being

Utilities

The `incr!` macro is just a little bit of syntactic sugar.

61a ⟨*util:macro* 61a⟩≡
```
(define-syntax-public incr!
  (syntax-rules ()
    ((incr! variable inc)
     (begin
       (set! variable (+ variable inc))
       variable))
    ((incr! variable)
     (incr! variable 1))))
```

61b ⟨*util:macro* 61a⟩+≡
```
(define-syntax-public decr!
  (syntax-rules ()
    ((decr! variable inc)
     (incr! variable (- inc)))
    ((decr! variable)
     (decr! variable 1))))
```

61c ⟨*util:macro* 61a⟩+≡
```
(define-syntax-public cons!
  (syntax-rules ()
    ((cons! elm list)
     (begin
       (set! list (cons elm list))
       list))))
```

File Layout

61d ⟨*file:block.scm* 61d⟩≡
```
(define-module (emacsy block)
  #:use-module (ice-9 optargs)
  #:use-module (oop goops)
  #:use-module (emacsy util))
```
⟨*block:macro* 57g⟩
⟨*block:class* 60d⟩
⟨*block:state* 57d⟩
⟨*block:procedure* 57c⟩
⟨*block:process* (never defined)⟩

Layout for tests.

61e ⟨*file:block-test.scm* 61e⟩≡
```
(use-modules (emacsy block)
             (oop goops))

(eval-when (compile load eval)
           ;; Some trickery so we can test private procedures.
           (module-use! (current-module) (resolve-module '(emacsy block))))
```
⟨*+ Test Preamble* (never defined)⟩
⟨*block:test* 57a⟩
⟨*+ Test Postscript* (never defined)⟩

5.5 KLECL Module

> A box without hinges, key, or lid,
> yet golden treasure inside is hid.
>
> The Hobbit
> J. R. R. Tolkien

We finally have all the pieces to properly build the KLECL. First, we have to accept input.

5.5.1 emacsy-event

The embedding application will handle the actual IO, but it passes events to Emacsy for processing which are stored in a queue.

63a ⟨*klecl:procedure* 63a⟩≡
```
(define-public (emacsy-event event)
  (enq! event-queue event))
```

63b ⟨*klecl:state* 63b⟩≡
```
(define-public event-queue (make-q))
```

This is a convenience procedure to enqueue a key event.

63c ⟨*klecl:procedure* 63a⟩+≡
```
(define*-public (emacsy-key-event char #:optional (modifier-keys '()))
  (emacsy-event (make <key-event>
                  #:modifier-keys modifier-keys
                  #:command-char char)))
```

63d ⟨*klecl:procedure* 63a⟩+≡
```
(define*-public (emacsy-mouse-event position button state
                                    #:optional (modifier-keys '()))
  (emacsy-event
   (make <mouse-event>
     #:position position #:button button
     #:state state #:modifier-keys modifier-keys)))
```

And mainly for testing purposes we also want to discard all input. Or there are cases where we want to unread an event and push it to the front of the queue rather than the rear.

63e ⟨*klecl:procedure* 63a⟩+≡
```
(define-public (emacsy-discard-input!)
  (while (not (q-empty? event-queue))
    (deq! event-queue)))

(define-public (emacsy-event-unread event)
  (q-push! event-queue event))
```

5.5.2 read-event

`read-event` is the lowest-level procedure for grabbing events. It will block if there are no events to read.

64 ⟨*klecl:procedure* 63a⟩+≡

```
#;(define*-public (read-event #:optional (prompt #f))
  (if prompt
      (message prompt))
  ;(if (q-empty? event-queue)
  (block-while (lambda () (q-empty? event-queue)) #t)
  (let ((event (deq! event-queue)))
    (run-hook read-event-hook event)
    event))

(define (raw-read-event prompt)
  (if prompt
      (message prompt))
  (let ((event (deq! event-queue)))
    (run-hook read-event-hook event)
    (emacsy-log-debug "RAW-READ-EVENT ~a~%" event)
    event))

(define*-public (read-event #:optional (prompt #f))
  (if emacsy-interactive?
      (if (q-empty? event-queue)
          (yield (lambda (resume)
                   (block-read-event prompt resume)))
          (raw-read-event prompt))
      ;; We're non-interactive. I need to read from stdin.

      (let ((input-string (read-line)))
        (if (and (eof-object? input-string)
                 (q-empty? event-queue))
            (throw 'read-event-eof)
            (begin
              (unless (eof-object? input-string)
                (map emacsy-event (kbd->events input-string)))
              (raw-read-event prompt))))))

(define reader-request-queue (make-q))

(define (block-read-event prompt resume)
  (format #t "block-read-event ~a~%" (list prompt resume))
  (enq! reader-request-queue (list prompt resume)))

(codefine (fulfill-read-requests)
   (while #t
     (format #t "fulfill-read-requests CHECK~%")
     (when (and (not (q-empty? event-queue))
                (not (q-empty? reader-request-queue)))
       (format #t "fulfill-read-requests DO~%")
       (match (deq! reader-request-queue)
         ((prompt resume)
          ;; Do I need to schedule this with the agenda to make it
```

```
            ;; behave properly?
            (resume (raw-read-event prompt)))))
        (wait)))

  (agenda-schedule fulfill-read-requests)
```

65a ⟨*klecl:state* 63b⟩+≡

```
  (define-public read-event-hook (make-hook 1))
  (define-public emacsy-interactive? #f)
```

65b ⟨*klecl:test* 65b⟩≡

```
  (define last-event #f)
  (codefine (test-read-event)
    (set! last-event (read-event)))
  ;(with-blockable (test-read-event))
  (agenda-schedule test-read-event)
  ;(check (blocking?) => #t)
  (set! emacsy-interactive? #t)
  (check last-event => #f)
  ;(block-tick)
  (update-agenda)
  (check last-event => #f)
  (emacsy-key-event #\a)

  (update-agenda)
  ;(check last-event => #f)
  ;(check (blocking?) => #f)
  (check (command-char last-event) => #\a)
  ;(clear-agenda)
```

5.5.3 read-key

Read key is slightly more high level than `read-event`. It may do a little processing of coalesing of events. For instance, down and up mouse events may be changed into click or drag events.

There's probably a better way of handling disparate classes of events use polymorphism!

66a ⟨*klecl:procedure* 63a⟩+≡

```
(define last-down-mouse-event #f)

(define*-public (read-key #:optional (prompt #f))
  (define* (new-mouse-state new-state event #:optional (event2 #f))
    (let ((e (make (if event2 <drag-mouse-event> <mouse-event>)
             #:modifier-keys (modifier-keys event)
             #:position (position event)
             #:button (button event)
             #:state new-state)))
      (if event2
          (slot-set! e 'rect (list (position event) (position event2))))
      e))
  ;; XXX Can this be refashioned into a nice handle-event method?
  ;; That would split up all these mouse key concerns.
  (let loop ((event (read-event prompt)))
    (if (is-a? event <dummy-event>)
        ;; Ignore it.
        (loop (read-event prompt))
        (if (down-mouse-event? event)
            (begin
              (set! last-down-mouse-event event)
              (loop (read-event prompt)))
            (if (and last-down-mouse-event
                     (down-mouse-event? last-down-mouse-event)
                     (up-mouse-event? event))
                (let ((new-event
                       (if (vector= (position last-down-mouse-event) (position event))
                           ;; Is it a click?
                           (new-mouse-state 'click event)
                           ;; Or a drag?
                           (new-mouse-state 'drag last-down-mouse-event event ))))
                  (set! last-down-mouse-event #f)
                  new-event)
                event)))))
```

66b ⟨*util:procedure* 66b⟩≡

```
(define-public (vector= a b)
  (assert (vector? a) (vector? b) report: "vector= ")
  (let ((len (vector-length a)))
   (and (= (vector-length a) (vector-length b))
        (let loop ((i 0))
          (if (>= i len)
              #t
              (if (= (vector-ref a i) (vector-ref b i))
               (loop (1+ i))
               #f))))))
```

5.5.4 read-key-sequence

`read-key-sequence` is at a higher level than `read-key`. It considers the currently active keymaps. If a sequence is defined in the keymap, it returns that event sequence. If it doesn't match a sequence yet, it continues to read more keys. If it cannot match any sequence, it returns that event sequence.

Consider using values to return multiple values.

67a ⟨*klecl:procedure* 63a⟩+≡

```
(define*-public (read-key-sequence
                 #:optional
                 (prompt #f)
                 #:key
                 (keymaps (default-klecl-maps)))
  (define (read-discrete-key)
    (let mini-loop ((event (read-key prompt)))
      (emacsy-log-trace "EVENT ~a~%" event)
      (if (discrete-event? event)
          event
          (mini-loop (read-key prompt)))))
  (let loop ((events (list (read-key prompt))))
    (let* ((keys (reverse (map (compose event->kbd canonize-event!) events)))
           (last-key (rcar keys)))
      ;; Do we have enough keys?
      (if (or
           ;; Does one of the keymaps points to a command (or is it the
           ;; quit key)?
           (or (quit-key? last-key keymaps)
               (any (lambda (keymap)
                      (lookup-key? keymap keys))
                    keymaps))
           ;; OR does none of the keymaps point to a command or keymap (or
           ;; is the quit key)?
           (not (or (quit-key? last-key keymaps)
                    (any (lambda (keymap)
                           (lookup-key? keymap keys #t))
                         keymaps))))
          ;; Yes. We have enough keys.
          (reverse events)
          ;; No.  Let's get some more.
          (loop (cons (read-discrete-key) events))))))
```

We also check all the maps for a quit key, typically defined as `C-g`.

67b ⟨*klecl:procedure* 63a⟩+≡

```
(define-public (quit-key? aKey keymaps)
  (define (quit-key?* key keymap)
    (let ((result (lookup-key keymap (list key))))
      (and (not (keymap? result)) (lookup-key-entry? result)
           (eq? 'keyboard-quit result))))
  (any (lambda (keymap) (quit-key?* aKey keymap)) keymaps))
```

67c ⟨*klecl:command* 67c⟩≡

```
(define-interactive (keyboard-quit)
  (message "Quit!")
  (throw 'quit-command))
```

Since we have no keymaps defined, `read-key-sequence` should quickly return any single key inputs.

68a ⟨*klecl:test* 65b⟩+≡
```
(define last-key-seq #f)
(codefine (read-key-sequence*)
  (set! last-key-seq #f)
  (set! last-key-seq (map command-char (read-key-sequence))))
;(with-blockable (read-key-sequence*))
(agenda-schedule read-key-sequence*)
(emacsy-key-event #\a)
;(block-tick)
(update-agenda)
(check last-key-seq => '(#\a))
(update-agenda)
(check last-key-seq => '(#\a))
```

However, if we add a keymap with only the sequence `a b c`, we will see that it'll behave differently.

68b ⟨*klecl:test* 65b⟩+≡
```
(define (no-command) #f)
(define test-keymap (make-keymap))
(set! default-klecl-maps (lambda () (list test-keymap)))
(define-key test-keymap "a b c" 'no-command)
;(with-blockable (read-key-sequence*))
(agenda-schedule read-key-sequence*)
(emacsy-key-event #\a)
;(block-tick)
(update-agenda)
(check last-key-seq => #f) ;; Not enough keys to return.
```

Let's test a sequence that is not in the keymap.

68c ⟨*klecl:test* 65b⟩+≡
```
(emacsy-key-event #\z)
(update-agenda)
(check last-key-seq => '(#\a #\z)) ;; No way "a z" is an actual key-sequence.
```

68d ⟨*klecl:test* 65b⟩+≡
```
;(with-blockable (read-key-sequence*))
(agenda-schedule read-key-sequence*)
(emacsy-key-event #\a)
(emacsy-key-event #\b)
;(block-tick)
(update-agenda)
(check last-key-seq => #f) ;; Not enough keys to return yet.
(emacsy-key-event #\c)
(update-agenda)
(check last-key-seq => '(#\a #\b #\c)) ;; Got it!
```

Let's test keyboard quitting.

69a ⟨*klecl:test* 65b⟩+≡
```
(define-key test-keymap "q" 'keyboard-quit)
;(with-blockable (read-key-sequence*))
(agenda-schedule read-key-sequence*)
(emacsy-key-event #\a)
;(block-tick)
(update-agenda)
(check last-key-seq => #f) ;; Not enough keys to return yet.
(emacsy-key-event #\q)
;(block-tick)
(update-agenda)
(check last-key-seq => '(#\a #\q)) ;; Got it!
```

Rename default-klecl-maps to current-active-maps.

69b ⟨*klecl:procedure* 63a⟩+≡
```
(define-public (default-klecl-maps)
  (list))
```

I find it convenient to begin emitting messages in case of error. However, I would like for there to be a clean separation between Emacsy and its KLECL such that someone may write a clean vim-y using it if they so chose. So this message will merely go to the stdout however, it will be redefined later.

69c ⟨*klecl:procedure* 63a⟩+≡
```
(define-public (message . args)
  (apply format #t args))
```

5.5.5 What is a command?

69d ⟨*util:procedure* 66b⟩+≡
```
(define*-public (with-backtrace* thunk #:optional (no-backtrace-for-keys '()))
  (with-throw-handler
   #t
   thunk
   (lambda (key . args)
     (when (not (memq key no-backtrace-for-keys))
       (emacsy-log-error
        "ERROR: Throw to key `~a' with args `~a'." key args)
       (backtrace)))))
```

We don't want to tie the embedders hands, so for any error output it ought to go through `emacsy-log-error`.

69e ⟨*util:procedure* 66b⟩+≡
```
(define-public (emacsy-log-error format-msg . args)
  (apply format (current-error-port) format-msg args)
  (newline (current-error-port)))

(define-public (emacsy-log-trace format-msg . args)
  (apply format (current-error-port) format-msg args)
  (newline (current-error-port)))
```

5.5.6 Command Loop

The command loop is a very important part of the KLECL. One note about loops is I sometimes write the body of a loop separate from the rest. Each iteration I call usually suffix with the name `-tick`.

70a ⟨*LOOP example* 70a⟩≡
```
(define (X-loop)
  (let loop ((count 0))
    (if (X-tick)
        (loop (1+ count)))))
```

With the command loop I've also adopted a prefix of `primitive-` which signifies that it does not do any error handling. The command loop sets up a fair amount of state.

70b ⟨*klecl:state* 63b⟩+≡
```
(define-public this-command-event #f)
(define-public last-command-event #f)

(define-public pre-command-hook (make-hook))
(define-public post-command-hook (make-hook))
```

70c ⟨*klecl:procedure* 63a⟩+≡
```
(define call-with-sigalrm
  (if (not (provided? 'posix))
      (lambda (thunk) (thunk))
      (lambda (thunk)
        (let ((handler #f))
          (dynamic-wind
            (lambda ()
              (set! handler
                    (sigaction SIGALRM
                      (lambda (sig)
                        #;(block-yield)
                        (scm-error 'signal #f "Alarm interrupt" '()
                                   (list sig))))))
            thunk
            (lambda ()
              (if handler
                  ;; restore Scheme handler, SIG_IGN or SIG_DFL.
                  (sigaction SIGALRM (car handler) (cdr handler))
                  ;; restore original C handler.
                  (sigaction SIGALRM #f))))))))
```

XXX Rename this to klec, for Key-Lookup-Execute-Command (KLEC)—just missing the loop component?

71a ⟨*klecl:procedure* 63a⟩+≡

```
(define*-public (primitive-command-tick #:optional
                                        (prompt #f)
                                        #:key
                                        (keymaps (default-klecl-maps))
                                        (undefined-command undefined-command))
  "We do one iteration of the command-loop without any error handling."
  (call-with-sigalrm
   (lambda ()
     ((@@ (ice-9 top-repl) call-with-sigint)
      (lambda ()
        (let* ((events (read-key-sequence prompt #:keymaps keymaps))
               (key-sequence (map event->kbd events))
               (keymap (find (lambda (k) (lookup-key? k key-sequence)) keymaps)))
          (set! emacsy-ran-undefined-command? #f)
          (if keymap
              (begin
                (set! last-command-event this-command-event)
                (set! this-command-event (rcar events))
                ;; The command hooks might need to go into the command module.
                (in-out
                 (run-hook pre-command-hook)
                 (call-interactively (lookup-key keymap key-sequence))
                 (run-hook post-command-hook)))
              ;; Maybe this should be done by an undefined-command command?
              ;; I doubt we want this command to be executed by the user, so
              ;; we're going to leave it as a procedure.
              (undefined-command key-sequence events))))))))
```

Uses `in-out` 95a.

71b ⟨*klecl:state* 63b⟩+≡

```
(define-public emacsy-ran-undefined-command? #f)
```

71c ⟨*klecl:procedure* 63a⟩+≡

```
(define* (undefined-command key-sequence events)
  (message "~a is undefined."
           (string-join key-sequence " "))
  (set! emacsy-ran-undefined-command? #t)
  (values #f 'no-such-command))
```

71d ⟨*util:state* 71d⟩≡

```
(define-public debug-on-error? #f)
```

72a ⟨*klecl:procedure* 63a⟩+≡

```
(define*-public (command-tick #:key (keymaps (default-klecl-maps)))
  "We do one iteration of command-tick and handle errors."

  (catch #t
    (lambda ()
      (if debug-on-error?
          (call-with-error-handling
           (lambda ()
             (primitive-command-tick #:keymaps keymaps))
           #:pass-keys
           ;; XXX what the hell is the story with all these quits?
           '(silent-quit quit-command-loop quit-command keyboard-quit))

          (with-backtrace* (lambda ()
                             (primitive-command-tick #:keymaps keymaps))
                           '(silent-quit quit-command-loop))))
    (lambda (key . args)
      (case key
        ((silent-quit)
         (emacsy-log-warning "GOT SILENT QUIT in command-tick\n"))
        ((quit-command-loop)
         (emacsy-log-warning "GOT QUIT-COMMAND-LOOP in command-tick\n")
         (apply throw key args))
        ((encoding-error)
         (emacsy-log-warning "ENCODING-ERROR '~a'" (pp-string event-queue)))
        (else
         (emacsy-log-error
                "command-tick: Uncaught throw to '~a: ~a\n" key args))))))
```

Let's define a convenience procedure to `pretty-print` to a string.

72b ⟨*util:procedure* 66b⟩+≡

```
(define-public (pp-string obj)
  (call-with-output-string (lambda (port) (pp obj port))))
```

72c ⟨*klecl:test* 65b⟩+≡

```
(define my-command-count 0)
(define-interactive (my-command)
  (incr! my-command-count))
(define-key test-keymap "c" 'my-command)
(emacsy-key-event #\c)
(check my-command-count => 0)
(agenda-schedule (colambda () (primitive-command-tick)))
(update-agenda)
;(with-blockable (primitive-command-tick))
(check my-command-count => 1)
```

Now let's write the command loop without any error handling. This seems a little messy with the continue predicate procedure being passed along. I'm not sure yet, how best to organize it.

73a ⟨*klecl:procedure* 63a⟩+≡
```
(define*-public (primitive-command-loop #:optional (continue-pred (const #t)))
  "We iterate with command-tick but we do not handle any errors."
  (with-fluids ((continue-command-loop? #t))
    (let loop ((continue? (call-with-values
                              primitive-command-tick
                            continue-pred)))
      (if (and (fluid-ref continue-command-loop?) continue?)
          (loop (call-with-values
                    primitive-command-tick
                  continue-pred))
          (decr! command-loop-count)))))
```

Each command loop is given a different number.

73b ⟨*klecl:state* 63b⟩+≡
```
(define command-loop-count 0)
```

Finally, here's our command loop with error handling.

73c ⟨*klecl:procedure* 63a⟩+≡
```
(define* (command-loop #:optional (continue-pred (const #t)))
  "We iterate with command-tick and handle errors."
  (catch #t
    (lambda ()
      (if debug-on-error?
          (call-with-error-handling
           (lambda ()
             (primitive-command-loop continue-pred))
           #:pass-keys
           '(silent-quit quit-command-loop quit-command keyboard-quit))

          (with-backtrace* (lambda ()
                             (primitive-command-loop continue-pred))
                           '(silent-quit quit-command-loop))))
    (lambda (key . args)
      (case key
        ((silent-quit)
         (emacsy-log-warning "GOT SILENT QUIT in command-loop"))
        ((quit-command-loop)
         (emacsy-log-warning "GOT QUIT-COMMAND-LOOP in command-loop")
         (apply throw key args))
        ((encoding-error)
         (emacsy-log-error "ENCODING-ERROR '~a'" (pp-string event-queue)))
        (else
         (emacsy-log-error
                "command-loop: Uncaught throw to '~a: ~a\n" key args))))))
```

We have finished the KLECL. Note that although we have used Emacs-like function names, we have not implemented the Emacs-like UI yet. We have not defined any default key bindings. I want to encourage people to explore different user interfaces based on the KLECL, and one can start from this part of the code. If one wanted to create a modal UI, one could use the `(emacsy klecl)` module and not have to worry about any "pollution" of Emacs-isms.

File Layout

74a ⟨*file:klecl.scm* 74a⟩≡

```
(define-module (emacsy klecl)
  #:use-module (ice-9 optargs)
  #:use-module (ice-9 match)
  #:use-module (ice-9 rdelim)
  #:use-module (ice-9 q)
  #:use-module (srfi srfi-1)
  #:use-module (rnrs io ports)
  #:use-module (oop goops)
  #:use-module (system repl error-handling)
  #:use-module (emacsy util)
  #:use-module (emacsy keymap)
  #:use-module (emacsy event)
  #:use-module (emacsy command)
  #:use-module (emacsy block)
  #:use-module (emacsy coroutine)
  #:use-module (emacsy agenda)
  #:export (command-loop)
  )
```

⟨*klecl:macro* (never defined)⟩
⟨*klecl:class* (never defined)⟩
⟨*klecl:state* 63b⟩
⟨*klecl:procedure* 63a⟩
⟨*klecl:command* 67c⟩
⟨*klecl:process* (never defined)⟩

Layout for tests.

74b ⟨*file:klecl-test.scm* 74b⟩≡

```
(use-modules (emacsy klecl)
             (emacsy event)
             (check)
             (oop goops))

(use-private-modules (emacsy klecl))
```

⟨*+ Test Preamble* (never defined)⟩
⟨*klecl:test* 65b⟩
⟨*+ Test Postscript* (never defined)⟩

5.6 Advice

> No enemy is worse than bad advice.
>
> Sophocles

Emacs has a facility to define "advice" these are pieces of code that run before, after, or around an already defined function. This article provides a good example.

75a ⟨*file:advice.scm* 75a⟩≡
```
(define-module (emacsy advice)
  #:use-module (srfi srfi-9)
  )
```

⟨*Record* 75b⟩

⟨*State* 76c⟩

⟨*Procedure* 76a⟩

How will this work? Before we try to make the macro, let's focus on building up the functions. We want to have a function that we can substitute for the original function which will have a number of before, after, and around pieces of advice that can be attached to it.

75b ⟨*Record* 75b⟩≡
```
(define-record-type <record-of-advice>
  (make-record-of-advice original before around after)
  record-of-advice?
  (original   advice-original)
  (before     advice-before    set-advice-before!)
  (around     advice-around    set-advice-around!)
  (after      advice-after     set-advice-after!))
```

75c ⟨*Record* 75b⟩+≡
```
(define-record-type <piece-of-advice>
  (make-piece-of-advice procedure name class priority flag)
  piece-of-advice?
  (procedure poa-procedure)
  (name      poa-name) ;; symbol not string
  (class     poa-class    set-poa-class!)
  (priority  poa-priority set-poa-priority!)
  (flag      poa-flag     set-poa-flag!))
```

76a ⟨*Procedure* 76a⟩≡
```
(define (make-advising-function advice)
  (lambda args
    (let ((around-advices (append (advice-around advice)
                                  (list (make-piece-of-advice
                                         (advice-original
                                          advice)
                                         'original
                                         'bottom
                                         0
                                         'activate))))
          (result #f))
      (define (my-next-advice)
        (if (null? around-advices)
            (throw 'next-advices-drained)
            (let ((next-one-around (car around-advices)))
              (set! around-advices (cdr around-advices))
              (apply (poa-procedure next-one-around) args))))
      ;; This could be done more cleanly.  For instance,
      ;; If one calls (next-advice) more than once,
      ;; they drain all the advice rather than calling
      ;; the same advice again, which is probably
      ;; the more correct behavior.

      (for-each (lambda (before)
                  (apply (poa-procedure before) args))
                (advice-before advice))

      (set! result (with-fluid* next-advice-func my-next-advice
                                (lambda ()
                                  (next-advice))))
      (for-each (lambda (after)
                  (apply (poa-procedure after) result args))
                (advice-after advice))
      result)))
```

76b ⟨*Procedure* 76a⟩+≡
```
(define (next-advice)
  (if (fluid-bound? next-advice-func)
      ((fluid-ref next-advice-func))
      (throw 'no-next-advice-bound)))
```

76c ⟨*State* 76c⟩≡
```
(define next-advice-func (make-fluid))
```

To test this functionality, we're going to make some counter procedures.

77a ⟨*advice:test* 77a⟩≡
```
(define (my-orig-func x)
  (+ x 1))

(define (make-counter)
  (let ((x 0))
    (lambda args
      (if (and (= (length args) 1) (eq? (car args) 'count))
          x
          (begin (set! x (+ x 1))
                 (car args))))))

(define a-before (make-counter))
```

Let's make an identity advice procedure. It does nothing, but it does wrap around the function.

77b ⟨*advice:test* 77a⟩+≡
```
(define advice (make-record-of-advice my-orig-func '() '() '()))

(define advised-func (make-advising-function advice))
(check (a-before 'count) => 0)
(check (my-orig-func 1) => 2)
(check (advised-func 1) => 2)
(check (a-before 'count) => 0)
```

Let's test this with the simple functionality of having a piece of before advice.

77c ⟨*advice:test* 77a⟩+≡
```
(define advice (make-record-of-advice my-orig-func (list (make-piece-of-advice a-before 'a-before 'be

(define advised-func (make-advising-function advice))
(check (a-before 'count) => 0)
(check (my-orig-func 1) => 2)
(check (advised-func 1) => 2)
(check (a-before 'count) => 1)
```

Let's check the after advice.

77d ⟨*advice:test* 77a⟩+≡
```
(define a-after (make-counter))
(define advice (make-record-of-advice my-orig-func '() '()
                                      (list (make-piece-of-advice a-after 'a-after 'after 0 'activate

(define advised-func (make-advising-function advice))
(check (a-after 'count) => 0)
(check (my-orig-func 1) => 2)
(check (advised-func 1) => 2)
(check (a-after 'count) => 1)
```

Let's check the after advice.

78a ⟨*advice:test* 77a⟩+≡
```
(define a-around (lambda args
                   (next-advice)
                   1))
(define advice (make-record-of-advice my-orig-func '() (list (make-piece-of-advice a-around 'a-around

(define advised-func (make-advising-function advice))
(check (my-orig-func 1) => 2)
(check (advised-func 1) => 1)
```

So the rudimentary elements of our advice facility works. Now, we want to make it so that we can alter the advice of an existing function. We'll add a property to the procedure, such that we can tell if it is an "advised" procedure.

78b ⟨*advice:procedure* 78b⟩≡
```
(define (advised? proc)
  (and (procedure? proc)
       (assq 'record-of-advice
             (procedure-properties proc))
       #t))
```

78c ⟨*advice:procedure* 78b⟩+≡
```
(define (make-advising-function* proc record-of-advice)
  (let ((advised-proc (make-advising-function proc record-of-advice)))
    (set-procedure-property! proc 'record-of-advice record-of-advice)))
```

78d ⟨*advice:procedure* 78b⟩+≡
```
(define (remove-advice! procedure name)
  (if (advised? procedure)
   (let ((roa (procedure-property procedure 'record-of-advice)))
     (for-each (lambda (class)
                 (match (getter-and-setter-for-poa class)
                   ((getter setter!)

                    (setter! (delete! name (getter roa) (lambda (name elem)
                                                          (eq? (poa-name elem))))))))
               '(before around after)))))
```

78e ⟨*advice:procedure* 78b⟩+≡
```
(define (getter-and-setter-for-poa poa-class)
  (cond ((eq? poa-class 'before)
         (cons advice-before set-advice-before!)
         (eq? poa-class 'around)
         (cons advice-around set-advice-around!)
         (eq? poa-class 'after)
         (cons advice-after set-advice-after!)
         (else
          (throw 'invalid-advice-class)))))
```

79a ⟨*advice:procedure* 78b⟩+≡

```
(define (add-advice! procedure piece-of-advice)
  "Add a piece-of-advice to the procedure.  Returns the advised
procedure."
  (define (sort-by! lst f)
    (sort! lst (lambda (a b)
                 (< (f a) (f b)))))
  (if (not (advised? procedure))
      ;; Procedure has never been advised.
      (add-advice (make-advising-function* procedure (make-record-of-advice procedure '() '() '())) pi
      ;; Add a new piece of advice.
      (begin
        (remove-advice! procedure (poa-name piece-of-advice))
        (let ((roa (procedure-property procedure 'record-of-advice)))
          (match (getter-setter-for-poa (poa-class piece-of-advice))
            ((getter setter!)
             (setter!
              roa
              (sort-by! (cons piece-of-advice (getter roa))
                        poa-priority)))))
        procedure)))
```

79b ⟨*file:advice-test.scm* 79b⟩≡

```
(use-modules (emacsy advice)
             (emacsy event)
             (emacsy klecl)
             (oop goops)
             (srfi srfi-11))

(eval-when (compile load eval)
           ;; Some trickery so we can test private procedures.
           (module-use! (current-module) (resolve-module '(emacsy advice))))
```

⟨+ *Test Preamble* (never defined)⟩
⟨*advice:test* 77a⟩
⟨+ *Test Postscript* (never defined)⟩

Chapter 6

Emacs-like Personality

We now want to take our KLECL and implement an Emacs-like UI on top of it. This will include buffers, keyboard macros, and a minibuffer.

6.1 Buffer Module

> And when you gaze long into an abyss the abyss also gazes into you.
>
> Beyond Good and Evil
> Friedrich Nietzsche

This should be moved out of the KLECL chapter.

A buffer in Emacs represents text, including its mode, local variables, etc. A Emacsy buffer is not necessarily text. It can be extended to hold whatever the host application is interested in. Emacs' concepts of buffer, window, and mode are directly analogous to the model, view, and controller respectively—the MVC pattern.

82a ⟨*buffer:class* 82a⟩≡
```
(define-class-public <buffer> ()
  (name #:init-keyword #:name)
  (keymap #:accessor local-keymap #:init-keyword #:keymap #:init-form (make-keymap))
  (locals #:accessor local-variables #:init-form '())
  (buffer-modified? #:accessor buffer-modified? #:init-value #f)
  (buffer-modified-tick #:accessor buffer-modified-tick #:init-value 0)
  (buffer-enter-hook #:accessor buffer-enter-hook #:init-form (make-hook 0))
  (buffer-exit-hook #:accessor buffer-exit-hook #:init-form (make-hook 0))
  (buffer-modes #:accessor buffer-modes #:init-form '()))
(export local-keymap local-variables buffer-enter-hook buffer-exit-hook before-buffer-change-hook afte
```

82b ⟨*buffer:state* 82b⟩≡
```
(define-variable before-buffer-change-hook (make-hook 1) "This hook is called prior to the buffer bei
(define-variable after-buffer-change-hook (make-hook 1) "This hook is called after to the buffer has
```

Buffer's have a name, and there is always a current buffer or it's false. Note that methods do not work as easily with optional arguments. It seems best to define each method with a different number of arguments as shown below.

82c ⟨*buffer:procedure* 82c⟩≡
```
(define-method-public (buffer-name)
  (buffer-name (current-buffer)))

(define-method-public (buffer-name (buffer <buffer>))
  (slot-ref buffer 'name))

(define-method-public (set-buffer-name! name)
  (set-buffer-name! name (current-buffer)))

(define-method-public (set-buffer-name! name (buffer <buffer>))
  (slot-set! buffer 'name name))

(define-method-public (buffer-modified?)
  (buffer-modified? (current-buffer)))

(define-method-public (buffer-modified-tick)
  (buffer-modified-tick (current-buffer)))

(define-method (write (obj <buffer>) port)
  (write (string-concatenate (list "#<buffer '" (buffer-name obj) "'>")) port))
```

82d ⟨*buffer:string* 82d⟩≡
```
(define-method-public (buffer-string)
  (buffer-string (current-buffer)))

(define-method-public (buffer-string (buffer <buffer>))
  (format #f "~a" buffer))
```

82e ⟨*buffer:test* 82e⟩≡
```
(define b (make <buffer> #:name "*test-buffer*"))
(check (buffer-name b) => "*test-buffer*")
(check (object->string b) => "\"#<buffer '*test-buffer*'>\"")
(check (current-buffer) => #f)
```

6.1.1 Emacs Compatibility

83 ⟨*buffer:procedure* 82c⟩+≡

```
(define-public (current-local-map)
  (local-keymap (current-buffer)))

(define-public (use-local-map keymap)
  (set! (local-keymap (current-buffer)) keymap))
```

6.1.2 Buffer List

The buffer module also keeps track of the live buffers and the current one.

Most Recently Used Stack

The buffers are kept in a most recently used stack that has the following operators: add!, remove!, contains?, recall!, and list.

84a ⟨*file:mru-stack.scm* 84a⟩≡

```
(define-module (emacsy mru-stack)
  #:use-module (ice-9 q)
  #:use-module (oop goops)
  #:use-module (emacsy util)
  #:export (<mru-stack>
            mru-add!
            mru-remove!
            mru-recall!
            mru-set!
            mru-ref
            mru-empty?
            mru-contains?
            mru-next!
            mru-prev!
            mru-list))
```

⟨*class* 84b⟩
⟨*procedure* 84c⟩

84b ⟨*class* 84b⟩≡

```
(define-class <mru-stack> ()
  (queue #:accessor q #:init-thunk (lambda () (make-q)))
  (index #:accessor index #:init-value 0))
```

84c ⟨*procedure* 84c⟩≡

```
(define-method (write (obj <mru-stack>) port)
;  (write (string-concatenate (list "#<mru-stack '" (buffer-name obj) "'>")) port)
  (format port "<mru-stack ~a>" (mru-list obj)))
```

85a ⟨*procedure* 84c⟩+≡

```
(define-method (mru-add! (s <mru-stack>) x)
  (q-push! (q s) x))
(define-method (mru-remove! (s <mru-stack>) x)
  (let ((orig-x (mru-ref s)))
    (q-remove! (q s) x)
    (if (not (eq? orig-x x))
        (mru-set! s orig-x))))
(define-method (mru-recall! (s <mru-stack>) x)
  (q-remove! (q s) x)
  (q-push! (q s) x)
  (set! (index s) 0)
  (mru-list s))
(define-method (mru-set! (s <mru-stack>) x)
  ;; Should this add the buffer if it's not already there? No.
  (if (mru-empty? s)
      #f
      (let ((i (member-ref x (mru-list s))))
        (if i
            (begin (set! (index s) i)
                   #t)
            (begin (mru-next! s)
                   #f)))))
(define-method (mru-ref (s <mru-stack>))
  (and (not (mru-empty? s))
       (list-ref (mru-list s) (index s))))
(define-method (mru-list (s <mru-stack>))
  (car (q s)))
(define-method (mru-empty? (s <mru-stack>))
  (q-empty? (q s)))
(define-method (mru-contains? (s <mru-stack>) x)
  (memq x (mru-list s)))
```

The order of the elements may not change yet the index may be moved around.

85b ⟨*procedure* 84c⟩+≡

```
(define-method (mru-next! (s <mru-stack>) count)
  (when (not (mru-empty?  s))
   (set! (index s)
         (modulo (+ (index s) count)
                 (length (mru-list s))))
   (mru-ref s)))
(define-method (mru-prev! (s <mru-stack>) count)
  (mru-next! s (- count)))
(define-method (mru-prev! (s <mru-stack>))
  (mru-prev! s 1))
(define-method (mru-next! (s <mru-stack>))
  (mru-next! s 1))
```

86a ⟨*file:mru-stack-test.scm* 86a⟩≡
```
(use-modules (emacsy mru-stack)
             (check))

(use-private-modules (emacsy mru-stack))
⟨test 86b⟩
(check-exit)
```

86b ⟨*test* 86b⟩≡
```
(define s (make <mru-stack>))
(mru-add! s 'a)
(mru-add! s 'b)
(mru-add! s 'c)
(check (mru-list s) => '(c b a))
(check (mru-recall! s 'a) => '(a c b))
(check (mru-ref s) => 'a)
(mru-next! s)
(check (mru-ref s) => 'c)
(mru-next! s)
(check (mru-ref s) => 'b)
(mru-next! s)
(check (mru-ref s) => 'a)
(mru-prev! s)
(check (mru-ref s) => 'b)
(check (mru-list s) => '(a c b))
(mru-remove! s 'c)
(check (mru-list s) => '(a b))
(check (mru-ref s) => 'b)
(mru-remove! s 'a)
(mru-remove! s 'b)
(check (mru-list s) => '())
(check (mru-ref s) => #f)
(mru-next! s)
(check (mru-ref s) => #f)
(mru-add! s 'a)
(mru-add! s 'b)
(mru-add! s 'c)
(check (mru-list s) => '(c b a))
(mru-remove! s 'c)
(check (mru-list s) => '(b a))
(check (mru-ref s) => 'b)
```

86c ⟨*buffer:state* 82b⟩+≡
```
(define-public buffer-stack (make <mru-stack>))
(define-public last-buffer #f)
```

86d ⟨*buffer:procedure* 82c⟩+≡
```
(define-public (buffer-list)
  (mru-list buffer-stack))
```

87a ⟨*buffer:procedure* 82c⟩+≡
```
(define-public (current-buffer)
  ;; Perhaps instead of returning #f for no buffer there should be an
  ;; immutable void-buffer class.
  (or aux-buffer
      (mru-ref buffer-stack)))
```

87b ⟨*buffer:procedure* 82c⟩+≡
```
(define-public (add-buffer! buffer)
  (mru-add! buffer-stack buffer))
```

87c ⟨*buffer:procedure* 82c⟩+≡
```
(define-public (remove-buffer! buffer)
  (mru-remove! buffer-stack buffer))

(define-interactive (next-buffer #:optional (incr 1))
  (mru-next! buffer-stack incr)
  (switch-to-buffer (mru-ref buffer-stack)))

(define-interactive (prev-buffer #:optional (incr 1))
  (next-buffer (- incr)))

(define-public (set-buffer! buffer)
  ;;(emacsy-log-debug "set-buffer! to ~a" buffer)
  (if (mru-set! buffer-stack buffer)
      (set! aux-buffer #f)
      (set! aux-buffer buffer)))

(define-interactive (kill-buffer #:optional (buffer (current-buffer)))
  (remove-buffer! buffer))

(define-interactive (other-buffer #:optional (count 1))
  (next-buffer count))
```

87d ⟨*buffer:state* 82b⟩+≡
```
(define-public aux-buffer #f)
```

`member-ref` returns the index of the element in the list if there is one.

87e ⟨*util:procedure* 87e⟩≡
```
(define-public (member-ref x list)
  (let ((sublist (member x list)))
    (if sublist
        (- (length list) (length sublist))
        #f)))
```

This is our primitive procedure for switching buffers. It does not handle any user interaction.

88a ⟨*buffer:procedure* 82c⟩+≡
```
(define (primitive-switch-to-buffer buffer)
  (emacsy-log-debug "Running exit hook for ~a" (current-buffer))
  (run-hook (buffer-exit-hook (current-buffer)))
  (set! last-buffer (current-buffer))
  (if (mru-contains? buffer-stack buffer)
      (begin
        (emacsy-log-debug "Recall buffer ~a" buffer)
        (mru-recall! buffer-stack buffer)
        (set! aux-buffer #f))
      (begin
        (emacsy-log-debug "Set buffer to ~a" buffer)
        (set-buffer! buffer)))
  (emacsy-log-debug "Running enter hook for ~a" (current-buffer))
  (run-hook (buffer-enter-hook (current-buffer)))
  (current-buffer))

(define-public switch-to-buffer primitive-switch-to-buffer)
```

88b ⟨*buffer:test* 82e⟩+≡
```
(add-buffer! b)
(check (buffer-name) => "*test-buffer*")
(remove-buffer! b)
(check (current-buffer) => #f)
```

Local Variables

88c ⟨*buffer:procedure* 82c⟩+≡
```
(define (local-var-ref symbol)
  (let ((result (assq symbol (local-variables (current-buffer)))))
    (if (pair? result)
     (cdr result)
     ;(variable-ref (make-undefined-variable))
     #f #;(throw 'no-such-local-variable symbol)))) ;; how can I throw an undefined value?

;; If buffers were in their own modules I could dynamically add variables
;; to their namespace.  Interesting idea.

(define (local-var-set! symbol value)
  (slot-set! (current-buffer)
             'locals
             (assq-set! (local-variables (current-buffer)) symbol value)))

(define-public local-var
               (make-procedure-with-setter local-var-ref local-var-set!))
```

6.1.3 Text Buffer

Our minibuffer and messages buffer require a text buffer.

89a ⟨*buffer:class* 82a⟩+≡
```
(define-class-public <text-buffer> (<buffer>)
  (gap-buffer #:accessor gap-buffer #:init-form (make-gap-buffer "")))
(export gap-buffer)
```

89b ⟨*buffer:procedure* 82c⟩+≡
```
(define-method-public (buffer-string (buffer <text-buffer>))
  (gb->string (gap-buffer buffer)))
```

Point

Now, let's implement the point procedures that control where the insertion point is within the buffer.

89c ⟨*buffer:procedure* 82c⟩+≡
```
(define-method-public (point)
  (point (current-buffer)))

(define-method-public (point (buffer <text-buffer>))
  (gb-point (gap-buffer buffer)))

(define-method-public (point-min)
  (point-min (current-buffer)))

(define-method-public (point-min (buffer <text-buffer>))
  (gb-point-min (gap-buffer buffer)))

(define-method-public (point-max)
  (point-max (current-buffer)))

(define-method-public (point-max (buffer <text-buffer>))
  (gb-point-max (gap-buffer buffer)))
```

Move the point around. It's very tempting to change the name from `goto-char` to `goto-point!` because `goto-char` is misleading. You don't actually go to a character, you go to a place between characters, a point.

Change goto-char to goto-point!

89d ⟨*buffer:procedure* 82c⟩+≡
```
(define-method-public (goto-char point)
  (goto-char point (current-buffer)))

(define-method-public (goto-char point (buffer <text-buffer>))
  (gb-goto-char (gap-buffer buffer) point))
;(define goto-point! goto-char)
```

Let's add the procedures `char-before` and `char-after` to inspect the characters before and after a point.

90 ⟨*buffer:procedure* 82c⟩+≡

```
;; XXX define-method needs to allow for the definition of optional arguments.
;; This is getting ridiculous.
(define-method-public (char-before)
  (char-before (point) (current-buffer)))

(define-method-public (char-before point)
  (char-before point (current-buffer)))

(define (gb-char-before gb point)
  ;; Avert thy eyes.
  (if (<= point (gb-point-min gb))
      #f
      (string-ref (gb->string gb) (- point 2))))

(define (gb-char-after gb point)
  ;; Avert thy eyes.
  (if (>= point (gb-point-max gb))
      #f
      (string-ref (gb->string gb) (- point 1))))

(define-method-public (char-before point (buffer <text-buffer>))
  (gb-char-before (gap-buffer buffer) point))

(define-method-public (char-after)
  (char-after (point) (current-buffer)))

(define-method-public (char-after point)
  (char-after point (current-buffer)))

(define-method-public (char-after point (buffer <text-buffer>))
  (gb-char-after (gap-buffer buffer) point))
```

There is documentation which suggests that `(ice-9 gap-buffer)` module has some nice regex search features. However, I can't find the implementation anywhere, so I implemented them pretty sloppily here. They should work find for a minibuffer, but probably shouldn't be used for anything bigger.

91 ⟨*buffer:procedure* 82c⟩+≡

```
(define*-public (gb-re-search-forward gb regex
                                      #:optional (bound #f) (no-error? #f) (repeat 1))
  ;; This could be done better in the gap-buffer.scm itself.
  (if (= repeat 0)
      (gb-point gb)
      (let* ((string (gb->string gb))
             (pt (gb-point gb))
             (match (regexp-exec regex string (- pt 1))))
        #;(format #t "match ~a ~%" match)
        (if match
            (begin
              (gb-goto-char gb (+ 1 (match:end match 0)))
              (gb-re-search-forward gb regex bound no-error? (1- repeat)))
            (if no-error?
                #f
                (scm-error 'no-match 'gb-re-search-forward

                           "No match found for regex '~a' in ~s after point ~a" (list regex string pt)

(define*-public (gb-re-search-backward gb regex
                                       #:optional (bound #f) (no-error? #f) (repeat 1))
  ;; This could be done better in the gap-buffer.scm itself.
  (if (= repeat 0)
      (gb-point gb)
      (let loop ((start-search 0)
                 (last-match-start #f))
       (let* ((string (gb->string gb))
              (pt (gb-point gb))
              (match (regexp-exec regex string start-search)))
         (define (my-error)
           (if no-error?
               #f
               (scm-error
                'no-match 'gb-re-search-forward
                "No match found for regex '~a' in ~s before point ~a"
                (list regex string pt) #f)))
         (define (finish)
           (if last-match-start
               (begin
                 (gb-goto-char gb (1+ last-match-start))
                 (gb-re-search-backward gb regex bound no-error? (1- repeat)))
               (my-error)))
         #;(format #t "match ~a ~%" match)
         (if match
             (if (< (match:start match 0) (1- pt))
                 ;; continue searching
                 (loop (match:end match 0) (match:start match 0))
                 (finish))
             (finish))))))
```

Some commands to move the point around and insert or delete characters.

92 ⟨*buffer:procedure* 82c⟩+≡

```
(define-interactive (kill-line #:optional (n 1))
  (gb-delete-char! (gap-buffer (current-buffer)) (- (point-max) (point))))

(define-interactive (delete-backward-char #:optional (n 1))
  (gb-delete-char! (gap-buffer (current-buffer)) (- n)))

(define-interactive (forward-delete-char #:optional (n 1))
  (gb-delete-char! (gap-buffer (current-buffer)) n))

(define-interactive (forward-char #:optional (n 1))
  (goto-char (+ (point) n)))

(define forward-word-regex (make-regexp "\\s*\\w+\\b"))
(define backward-word-regex (make-regexp "\\b\\w+\\s*"))

;; XXX where is gb-re-search-forward defined?  It has documentation.
;; but no implementation?
;; http://gnuvola.org/software/guile/doc/Gap-Buffer.html
(define-interactive (forward-word #:optional (n 1))
  (gb-re-search-forward (gap-buffer (current-buffer)) forward-word-regex #f #t n))

(define-interactive (backward-word #:optional (n 1))
  (gb-re-search-backward (gap-buffer (current-buffer)) backward-word-regex #f #t n))

(define-interactive (move-beginning-of-line #:optional (n 1))
  ;(gb-beginning-of-line (gap-buffer (current-buffer)) n)
  (goto-char (point-min)))

(define-interactive (move-end-of-line #:optional (n 1))
  ;(gb-beginning-of-line (gap-buffer (current-buffer)) n)
  (goto-char (point-max)))

(define-interactive (backward-char #:optional (n 1))
  (forward-char (- n)))
```

Let's test this regex search in a gap buffer.

93a ⟨*buffer:test* 82e⟩+≡

```
(define c (make <text-buffer> #:name "*test-regex*"))
(add-buffer! c)
(check (current-buffer) => c)
(check (buffer-modified?) => #f)
(check (buffer-modified-tick) => 0)
(insert "hellos these ard words!")
(check (buffer-modified?) => #t)
(check (buffer-modified-tick) => 1)
;;       1    7     13  17
(check (point) => (point-max))
(check (point-min) => 1)
(goto-char (point-min))
(check (gb-char-after (gap-buffer c) 1) => #\h)
(check (gb-char-before (gap-buffer c) 1) => #f)
(check (point) => 1)
(check (forward-word) => 7)
(check (point) => 7)
(check (char-before) => #\s)
(check (char-after) => #\space)

(check (forward-word 2) => 17)
(check (char-before) => #\d)
(check (char-after) => #\space)
(check (backward-word) => 14)
(check (char-before) => #\space)
(check (char-after) => #\a)

;;#(!sdrow dra eseht solleh (17 . 24))
;;  1      8   12    18
;; is               ^      ^
;; goto       ^
;; was      ^
```

Finally, we can insert text.

93b ⟨*buffer:procedure* 82c⟩+≡

```
(define*-public (insert #:rest args)
  (and (current-buffer)
   (if (null? args)
       0
       (let ((arg (car args)))
         (run-hook before-buffer-change-hook (current-buffer))
         (cond
          ((string? arg)
           (gb-insert-string! (gap-buffer (current-buffer)) arg))
          ((char? arg)
           (gb-insert-char! (gap-buffer (current-buffer)) arg))
          (else #f))
         (set! (buffer-modified? (current-buffer)) #t)
         (incr! (buffer-modified-tick (current-buffer)))
         (run-hook after-buffer-change-hook (current-buffer))))))
```

94a ⟨*buffer:procedure* 82c⟩+≡
```
(define-interactive (self-insert-command #:optional (n 1))
  (if (< n 1)
      ;; We're done.
      #f
      (let* ((event this-command-event))
        ;; Do I have to do anything for shifted characters?
        (insert (command-char event)))))
```

A convenience macro to work with a given buffer.

94b ⟨*buffer:macro* 94b⟩≡
```
(define-syntax-public with-buffer
  (syntax-rules ()
    ((with-buffer buffer e ...)
     (let ((old-buffer (current-buffer)))
       (in-out-guard
        (lambda () (set-buffer! buffer))
        (lambda () e ...)
        (lambda () (set-buffer! old-buffer)))))))
```
Uses in-out 95a.

This macro requires a procedure `in-out-guard` defined in the `util` module.

94c ⟨*util:procedure* 87e⟩+≡
```
(define*-public (in-out-guard in thunk out #:optional (pass-keys '(quit quit-command)))
  (run-thunk in)
  ;if debug-on-error?
  ;; Don't run this as robustly so that we can debug the errors
  ;; more easily.
  #;
  (receive (result . my-values) (run-thunk thunk)
  (run-thunk out)
  (apply values result my-values))

  (receive (result . my-values)
      (catch #t
        (if debug-on-error?
            (lambda ()
              (call-with-error-handling thunk #:pass-keys pass-keys))
            thunk)
        (lambda (key . args)
          (run-thunk out)
          (apply throw key args)))
    (run-thunk out)
    (apply values result my-values)))

;; Make code a little more obvious.
(define-public (run-thunk t)
  (t))
```
Uses in-out 95a.

95a ⟨*util:macro* 95a⟩≡
```
(define-syntax-public in-out
  (syntax-rules ()
    ((in-out in thunk out)
     (in-out-guard (lambda () in)
                   (lambda () thunk)
                   (lambda () out)))
    ((in-out in thunk out pass-keys)
     (in-out-guard (lambda () in)
                   (lambda () thunk)
                   (lambda () out)
                   pass-keys))))
```
Defines:
`in-out`, used in chunks 71a, 94, 95, 101, and 123c.

File Layout

95b ⟨*file:buffer.scm* 95b⟩≡
```
(define-module (emacsy buffer)
  #:use-module (ice-9 optargs)
  #:use-module (ice-9 q)
  #:use-module (ice-9 gap-buffer)
  #:use-module (ice-9 receive)
  #:use-module (ice-9 regex)
  #:use-module (srfi srfi-26)
  #:use-module (string completion)
  #:use-module (oop goops)
  #:use-module (emacsy util)
  #:use-module (emacsy mru-stack)
  #:use-module (emacsy self-doc)
  #:use-module (emacsy event)
  #:use-module (emacsy keymap)
  #:use-module (emacsy command)
  #:use-module (emacsy klecl)
  #:use-module (emacsy mode)
  #:use-module (rnrs base))
⟨buffer:macro 94b⟩
⟨buffer:class 82a⟩
⟨buffer:state 82b⟩
⟨buffer:procedure 82c⟩
⟨buffer:process (never defined)⟩
```

Layout for tests.

96 ⟨*file:buffer-test.scm* 96⟩≡

```
(use-modules (check)
             (emacsy mru-stack)
             (emacsy buffer)
             (emacsy command)
             (emacsy event)
             (emacsy keymap)
             (oop goops)
             (rnrs base))

(use-private-modules (emacsy buffer))
```

⟨+ *Test Preamble* (never defined)⟩
⟨*buffer:test* 82e⟩
⟨+ *Test Postscript* (never defined)⟩

6.2 Minibuffer

...

...

The minibuffer provides a rich interactive textual input system. It offers `TAB` completion and history. The implementation of it inherits from the `<text-buffer>`.

97a ⟨*minibuffer:class* 97a⟩≡
```
(define-class-public <minibuffer> (<text-buffer>)
  (prompt #:accessor minibuffer-prompt #:init-form "")
  (message #:accessor minibuffer-message-string #:init-form ""))
```

We define a keymap with all the typical self-insert-commands that would be expected in an editable buffer.

This should probably be defined in the buffer module since it is general.

97b ⟨*minibuffer:state* 97b⟩≡
```
(define-public minibuffer-local-map
  (let ((keymap (make-keymap)))
    (char-set-for-each
     (lambda (c)
       (let ((event (make <key-event>
                      #:command-char c)))
         (define-key keymap (list (event->kbd event))
           'self-insert-command)))
     (char-set-delete
      (char-set-intersection char-set:ascii char-set:printing)
      #\vtab #\page #\space #\nul))
    keymap))
```

We want to be able to move around the buffer as well.

97c ⟨*minibuffer:keymap* 97c⟩≡
```
(define-key minibuffer-local-map "C-f" 'forward-char)
(define-key minibuffer-local-map "M-f" 'forward-word)
(define-key minibuffer-local-map "C-b" 'backward-char)
(define-key minibuffer-local-map "M-b" 'backward-word)
(define-key minibuffer-local-map "DEL" 'delete-backward-char)
(define-key minibuffer-local-map "SPC" 'self-insert-command)
(define-key minibuffer-local-map "-"   'self-insert-command)
(define-key minibuffer-local-map "C-a" 'move-beginning-of-line)
(define-key minibuffer-local-map "C-e" 'move-end-of-line)
(define-key minibuffer-local-map "C-k" 'kill-line)
(define-key minibuffer-local-map "C-d" 'forward-delete-char)
(define-key minibuffer-local-map "RET" 'exit-minibuffer)
(define-key minibuffer-local-map "M-n" 'next-history-element)
(define-key minibuffer-local-map "M-p" 'previous-history-element)
(define-key minibuffer-local-map "C-g" 'keyboard-quit)
```

We instantiate the `<minibuffer>` class into the global variable `minibuffer`.

97d ⟨*minibuffer:state* 97b⟩+≡
```
(define-public minibuffer
  (make <minibuffer> #:keymap minibuffer-local-map #:name "*minibuffer-1*"))
```

Whenever the minibuffer is being used, we want to show it instead of the echo area, so we add some hooks.

98a ⟨*minibuffer:process* 98a⟩≡
```
(add-hook! (buffer-enter-hook minibuffer)
           (lambda ()
             (emacsy-log-debug "Enter minibuffer!\n")
             (set! emacsy-display-minibuffer? #t)))

(add-hook! (buffer-exit-hook  minibuffer)
           (lambda ()
             (emacsy-log-debug "Exit minibuffer!\n")
             (set! emacsy-display-minibuffer? #f)))
```

98b ⟨*util:procedure* 98b⟩≡
```
(define-public (emacsy-log-debug format-msg . args)
  (apply format (current-error-port) format-msg args)
  (newline (current-error-port)))
```

98c ⟨*minibuffer:state* 97b⟩+≡
```
(define-public emacsy-display-minibuffer? #f) ;; or the echo area
```

The minibuffer has a prompt, but we want it to behave generally like any other text buffer. So let's implement the procedures: `buffer-string`, `point`, `point-min`, `point-max`, and `goto-char`.

When we show the minibuffer, we'll show the prompt, the contents (user editable), and the minibuffer-message if applicable.

98d ⟨*minibuffer:procedure* 98d⟩≡
```
(define-method (buffer-string (buffer <minibuffer>))
  (string-concatenate (list
                       (minibuffer-prompt buffer)
                       (minibuffer-contents buffer)
                       (minibuffer-message-string buffer))))
```

98e ⟨*minibuffer:procedure* 98d⟩+≡
```
(define*-public (minibuffer-contents #:optional (buffer minibuffer))
  (gb->string (gap-buffer buffer)))

(define*-public (delete-minibuffer-contents #:optional (buffer minibuffer))
  (gb-erase! (gap-buffer buffer)))
```

For the point methods, we're going to make `(goto-char 1)` the beginning of the prompt, but `(point-min)` where the user editable content starts. Basically, it should be as though it were a regular buffer that has been narrowed.

98f ⟨*minibuffer:procedure* 98d⟩+≡
```
(define-method (point-min (buffer <minibuffer>))
  (+ (next-method) (string-length (minibuffer-prompt buffer))))

(define-method (point (buffer <minibuffer>))
  (+ (next-method) (string-length (minibuffer-prompt buffer))))

(define-method (point-max (buffer <minibuffer>))
  (+ (next-method) (string-length (minibuffer-prompt buffer))))
```

If the prompt changes, the point should be adjusted manualy.

For `goto-char` we just undo that thing.

99a ⟨*minibuffer:procedure* 98d⟩+≡
```
(define-method (goto-char point (buffer <minibuffer>))
  (gb-goto-char (gap-buffer buffer)
                (- point (string-length (minibuffer-prompt buffer)))))
```

99b ⟨*minibuffer:test* 99b⟩≡
```
(check (buffer-string minibuffer) => "")
(check (point-min minibuffer) => 1)
(set! (minibuffer-prompt minibuffer) "What? ")
(check (buffer-string minibuffer) => "What? ")
(check (point-min minibuffer) => 7)
(with-buffer minibuffer
             (insert "Nothing."))
(check (buffer-string minibuffer) => "What? Nothing.")
```

One can add a message to the minibuffer that can act as an interactive help or show possible completions. The message will only last until the next command is executed.

99c ⟨*minibuffer:procedure* 98d⟩+≡
```
(define* (seconds->ticks seconds #:optional (default-ticks #f))
  "Converts seconds to number of ticks, if such a conversion is
available. Otherwise returns default-ticks."
  (if ticks-per-second
      (* seconds ticks-per-second)
      default-ticks))

(define-public (minibuffer-message string . args)
  (set! (minibuffer-message-string minibuffer)
        (apply format #f string args))
  (incr! minibuffer-message-modified-tick)
  (agenda-schedule (let ((my-tick minibuffer-message-modified-tick))
                     (lambda ()
                       (when (= my-tick minibuffer-message-modified-tick)
                         (set! (minibuffer-message-string minibuffer) ""))))
                       (seconds->ticks minibuffer-message-timeout 1)))
```

99d ⟨*minibuffer:state* 97b⟩+≡
```
(define-public minibuffer-message-timeout 5)
(define-public ticks-per-second #f)
(define minibuffer-message-modified-tick 0)
```

100 ⟨*minibuffer:test* 99b⟩+≡

```
(set! default-klecl-maps (lambda () (list minibuffer-local-map)))
(set-buffer! minibuffer)
(delete-minibuffer-contents minibuffer)
(check (buffer-string minibuffer) => "What? ")
(insert "A")
(agenda-schedule (colambda ()
                   (minibuffer-message " [Huh?]")))
#;(with-blockable
  (minibuffer-message " [Huh?]"))
(update-agenda)
(check (buffer-string minibuffer) => "What? A [Huh?]")
(update-agenda)
(check (buffer-string minibuffer) => "What? A")
(emacsy-key-event #\c)
(agenda-schedule (colambda () (command-tick)))
(update-agenda)
(check (buffer-string minibuffer) => "What? Ac")
;(emacsy-key-event #\a)
;(block-tick)
;(check (buffer-string minibuffer) => "What? Aa")
```

6.2.1 read-from-minibuffer

101 ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public (read-from-minibuffer prompt #:optional
                                      (initial-contents #f)
                                      #:key
                                      (read #f)
                                      (keymap minibuffer-local-map)
                                      (history (what-command-am-i?)))
  "history can be #f, a symbol, or a <cursor-list>."
  (define (read-from-minibuffer-internal prompt read)
    (when minibuffer-reading?
      (minibuffer-message
       " [Command attempted to use minibuffer while in minibuffer.]")
      (throw 'quit-command 'already-in-minibuffer))
    (when history
      (cond
       ((symbol? history)
        (let ((entry (hashq-ref history-symbol-map history #f)))
          (unless entry
            (set! entry (make-history))
            (hashq-set! history-symbol-map history entry))
          (set! minibuffer-history entry)))
       ((cursor-list? history)
          (set! minibuffer-history history))
       (else
        (scm-error 'invalid-argument "read-from-minibuffer" "Expecting #f, a symbol, or a <cursor-list
    (history-insert! minibuffer-history "")
   (emacsy-log-debug "Switching to minibuffer now.")
   (switch-to-buffer minibuffer)
   (delete-minibuffer-contents minibuffer)
   (with-buffer minibuffer
    (when initial-contents
      (insert initial-contents))
    (goto-char (point-min)))
   (set! (minibuffer-prompt minibuffer) (or prompt ""))
   (in-out
    (set! minibuffer-reading? #t)
    (let ((canceled? #f))
      (catch
        'quit-command
        (lambda ()
          (while minibuffer-reading?
            (primitive-command-tick)))
        (lambda (key . args)
          (emacsy-log-debug "MINIBUFFER CANCELED!\n")
          (set! canceled? #t)))
      (if canceled?
          (begin
            (if (eq? (current-buffer) minibuffer)
                (switch-to-buffer last-buffer))
            (throw 'quit-command 'quit-read-from-minibuffer))
          (begin
            (history-set! minibuffer-history (minibuffer-contents))
```

```
            (cursor-right! minibuffer-history)
            (minibuffer-contents))))
      (set! minibuffer-reading? #f)))

  (let ((original-keymap #f)
        (original-history #f))
    (in-out
     (begin (set! original-keymap (local-keymap minibuffer))
            (set! (local-keymap minibuffer) keymap)
            (set! original-history minibuffer-history))
     (read-from-minibuffer-internal prompt read)
     (begin
       (set! (local-keymap minibuffer) original-keymap)
       (set! minibuffer-history original-history))
     '(quit-command keyboard-quit))))
```

Uses in-out 95a.

102a ⟨*minibuffer:state* 97b⟩+≡

```
(define minibuffer-reading? #f)
```

102b ⟨*minibuffer:command* 102b⟩≡

```
(define-interactive (exit-minibuffer)
  (set! minibuffer-reading? #f)
  (switch-to-buffer last-buffer))
```

Test regular input to minibuffer.

102c ⟨*minibuffer:test* 99b⟩+≡

```
(emacsy-discard-input!)
(emacsy-key-event #\a)
(emacsy-key-event #\cr)
(check (read-from-minibuffer "What? ") => "a")
```

Test quitting the minibuffer.

102d ⟨*minibuffer:test* 99b⟩+≡

```
(emacsy-discard-input!)
(emacsy-key-event #\a)
(emacsy-key-event #\g '(control))
;(with-backtrace* (read-from-minibuffer "What?1 "))
;(set! emacsy-interactive? #f)
(check-throw ((colambda () (read-from-minibuffer "What?1 "))) => 'quit-command)
;((colambda () (read-from-minibuffer "What?1 ")))
;; Displaying the "Quit!" message causes a pause.
;(check-throw (update-agenda) => 'quit-command)
;(set! emacsy-interactive? #t)
```

Test retaining history in the minibuffer.

103a ⟨*minibuffer:test* 99b⟩+≡
```
(emacsy-discard-input!)
(emacsy-key-event #\h)
(emacsy-key-event #\i)
(emacsy-key-event #\cr)
(emacsy-key-event #\b)
(emacsy-key-event #\y)
(emacsy-key-event #\e)
(emacsy-key-event #\cr)
;(with-backtrace* (read-from-minibuffer "What?1 "))
(let ((h (make-history '())))
  (check (read-from-minibuffer "What?3 " #:history h) => "hi")
  (check (cursor-list->list h) => '("hi"))
  (check (read-from-minibuffer "What?4 " #:history h) => "bye")
  (check (cursor-list->list h) => '("hi" "bye"))
  )
```

Test accessing history in the minibuffer.

103b ⟨*minibuffer:test* 99b⟩+≡
```
(emacsy-discard-input!)
(emacsy-key-event #\p '(meta))
(emacsy-key-event #\cr)
(emacsy-key-event #\1)
(emacsy-key-event #\cr)
;(with-backtrace* (read-from-minibuffer "What?1 "))
(let ((h (make-history '("hi"))))
  (check (cursor-list->list h) => '("hi"))
  (check (read-from-minibuffer "What?3 " #:history h) => "hi")
  (check (cursor-list->list h) => '("hi" ""))
  (check (read-from-minibuffer "What?5 " #:history h) => "1")
  (check (cursor-list->list h) => '("hi" "1" ""))
)
```

Test accessing history in the minibuffer using a symbol.

103c ⟨*minibuffer:test* 99b⟩+≡
```
(emacsy-discard-input!)
(emacsy-key-event #\1)
(emacsy-key-event #\cr)
(emacsy-key-event #\p '(meta))
(emacsy-key-event #\cr)
(emacsy-key-event #\p '(meta))
(emacsy-key-event #\cr)
;(with-backtrace* (read-from-minibuffer "What?1 "))
(let ((h (make-history '("hi"))))
  (check (read-from-minibuffer "What?6 " #:history 'h1) => "1")
  (check (read-from-minibuffer "What?7 " #:history 'h1) => "1")

  (check (read-from-minibuffer "What?6 " #:history 'h2) => "")
)
```

6.2.2 Tab Completion

We want to offer string completion similar to Emacs.

104a ⟨*minibuffer:test* 99b⟩+≡
```
(check (try-completion "f" (list "foo" "foobar" "barfoo")) => "foo")
(check (try-completion "b" (list "foo" "foobar" "barfoo")) => "barfoo")

;; Try against readline completer
(check (try-completion "f" (make-completion-function (list "foo" "foobar" "barfoo"))) => "foo")
(check (try-completion "b" (make-completion-function (list "foo" "foobar" "barfoo"))) => "barfoo")
```

It can also work with a procedure.

104b ⟨*minibuffer:test* 99b⟩+≡
```
(check (try-completion "f" (lambda (string predicate all?) (if (string=? string "f") "blah" "huh"))) =
(check (try-completion "w" (lambda (string predicate all?) (if (string=? string "f") "blah" "huh"))) =
```

104c ⟨*minibuffer:procedure* 98d⟩+≡
```
(define (readline-completer->stream completer string)
  (define iter
    (stream-lambda (f)
                   (let ((result (f)))
                     (if (stream-null? result)
                         result
                         (stream-cons result (iter f))))))
  (let ((first (completer string #f)))
    (if first
        (stream-cons
         first
         (iter
          (lambda ()
            (let ((result (completer string #t)))
              (if result
                  result
                  stream-null)))))
        stream-null)))
```

104d ⟨*minibuffer:test* 99b⟩+≡
```
(check (sort! (stream->list (readline-completer->stream command-completion-function "")) string<?) =>
```

104e ⟨*minibuffer:procedure* 98d⟩+≡
```
(define (readline-completer? proc)
  (let ((req (arity:nreq (car (program-arities proc)))))
    (= req 2)))
```

105a ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public
  (try-completion string collection #:optional (predicate (const #t)))

  (if (procedure? collection)
      (if (readline-completer? collection)
          (try-completion
           string
           (stream->list (readline-completer->stream collection string))
           predicate)
          (collection string predicate #f))
      (let ((completer (collection->completer collection predicate)))
        (receive (completions expansion exact? unique?)
            (complete completer string)
          expansion))))
```

105b ⟨*minibuffer:test* 99b⟩+≡

```
(check (all-completions "f" (list "foo" "foobar" "barfoo")) => (list "foo" "foobar"))
(check (all-completions "b" (list "foo" "foobar" "barfoo")) => (list "barfoo"))

(check (all-completions "f" (make-completion-function (list "foo" "foobar" "barfoo"))) => (list "foo"
(check (all-completions "b" (make-completion-function (list "foo" "foobar" "barfoo"))) => (list "barf
```

105c ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public
  (all-completions string collection #:optional (predicate (const #t)))
  (if (procedure? collection)
      (if (readline-completer? collection)
          (all-completions
           string
           (stream->list
            (readline-completer->stream collection string))
           predicate)
          (collection string predicate #t))
   (let ((completer (collection->completer collection predicate)))
     (receive (completions expansion exact? unique?)
         (complete completer string)
       completions))))
```

105d ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public
  (collection->completer collection #:optional (predicate (const #t)))
  (if (is-a? collection <string-completer>)
      collection

      (let ((completer (make <string-completer>)))
        (add-strings! completer (filter predicate collection))
        completer)))
```

106a ⟨*minibuffer:command* 102b⟩+≡

```
(define-interactive (minibuffer-complete)
  (with-buffer
   minibuffer
   (let* ((contents (substring (minibuffer-contents) 0 (- (point) (point-min))))
          (expansion (try-completion
                      contents
                      (fluid-ref minibuffer-completion-table)
                      (fluid-ref minibuffer-completion-predicate)))
          (completions (all-completions
                        contents
                        (fluid-ref minibuffer-completion-table)
                        (fluid-ref minibuffer-completion-predicate))))
     (format #t "contents = ~a, expansion = ~a, completions = ~a ~%" contents expansion completions)
     (while (not (= (point) (point-min)))
       (delete-backward-char))
                                       ;(goto-char (point-min))
                                       ;(kill-line)
     (insert expansion)
     (cond
      ((= 0 (length completions))
       (minibuffer-message " [No match]"))
      ((= 1 (length completions))
       (minibuffer-message " [Sole completion]"))
      ((> (length completions) 1)
       (minibuffer-message
        (string-concatenate
         (list "{"
               (string-join (rotate-list completions *nth-match*) " | ")
               "}")))))))))
```

106b ⟨*minibuffer:state* 97b⟩+≡

```
(define *nth-match* 0)
```

106c ⟨*minibuffer:command* 102b⟩+≡

```
(define-interactive (next-match)
  (incr! *nth-match*)
  (minibuffer-complete))

(define-interactive (previous-match)
  (decr! *nth-match*)
  (minibuffer-complete))
```

106d ⟨*minibuffer:procedure* 98d⟩+≡

```
(define (rotate-list lst nth)
  (let* ((nth (modulo nth (length lst)))
         (tail (list-tail lst nth))
         (head (list-head lst nth)))
    (append tail head)))
```

107a ⟨*minibuffer:command* 102b⟩+≡

```
(define-interactive (minibuffer-complete-word)
  ;; This should only complete a word.
  (minibuffer-complete))

(define-interactive (minibuffer-completion-help)
  ;; This should only complete a word.
  (message "minibuffer-complete-help NYI")
  #f)
```

107b ⟨*minibuffer:keymap* 97c⟩+≡

```
(define minibuffer-local-completion-map
  (let ((kmap (make-keymap minibuffer-local-map)))
    ;(define-key kmap "SPC" 'minibuffer-complete-word)
    (define-key kmap "TAB" 'minibuffer-complete)
    (define-key kmap "?"   'minibuffer-completion-help)
    (define-key kmap "C-s" 'next-match)
    (define-key kmap "C-r" 'previous-match)))
```

107c ⟨*minibuffer:state* 97b⟩+≡

```
(define minibuffer-completion-table (make-fluid '()))
(define minibuffer-completion-predicate (make-fluid (const #t)))
(define minibuffer-completion-confirm #f)
(define minibuffer-completion-exit-commands '())
```

107d ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public
  (completing-read prompt collection
                   #:key
                   (predicate (const #t))
                   (require-match? #f)
                   (initial-input #f)
                   (history (what-command-am-i?))
                   (to-string #f))

;; XXX implement require-match?
  (define (completing-read* collection*)
    (with-fluids ((minibuffer-completion-table collection*)
                  (minibuffer-completion-predicate predicate))
      (read-from-minibuffer prompt initial-input
                            #:keymap minibuffer-local-completion-map
                            #:history history)))
  (cond
   (to-string
    (receive (to-string* from-string*) (object-tracker to-string)
      (from-string* (completing-read* (map to-string* collection)))))
   (else
    (completing-read* collection))))
```

6.2.3 Filename Lookup

New

We can do filename lookups by using the readline tab completion facilities.

108 ⟨*minibuffer:procedure* 98d⟩+≡

```
(define-public (apropos-module rgx module)
  "Return a list of accessible variable names for a given module."
  (apropos-fold (lambda (module name var data)
                  (cons name data))
                '()
                rgx
                (apropos-fold-accessible module)))

(define-public command-completion-function
  (let ((completions '()))
    (lambda (text cont?)
      (if (not cont?)
          (set! completions
                (map symbol->string
                     (apropos-module
                      (string-append "^" (regexp-quote text))
                      (module-command-interface (resolve-module '(emacsy core)))))))
      (if (null? completions)
          #f
          (let ((retval (car completions)))
            (begin (set! completions (cdr completions))
                   retval))))))
```

Old

We want to be able to look up filenames.

109 ⟨*minibuffer:procedure* 98d⟩+≡

```
(define (files-in-dir dirname)
  (let ((dir (opendir dirname))
        (filenames '()))
    (let loop ((filename (readdir dir)))
      (when (not (eof-object? filename))
        (cons! filename filenames)
        (loop (readdir dir))))
    (closedir dir)
    (sort-filenames filenames)))

(define (make-filename dir name)
  (format #f "~a~a" (if (slash-suffix? dir)
                        dir
                        (format #f "~a/" dir)) name))

(define (sort-filenames lst)
  (sort lst string<))

(define (files-in-parent-dir filename)

  (let* ((dir (my-dirname filename))
         ;(base (basename filename))
         (filenames (files-in-dir dir)))
    (sort-filenames
     (if (or (string-prefix? dir filename)
             (string-prefix? filename dir)) ;; the directory is a prefix of
         ;; the string--good!
         (map (compose canonize-filename (cut make-filename dir <>)) filenames)
         (map canonize-filename filenames)))))

(define (directory? filename)
  (and (access? filename F_OK)
       (eq? 'directory (stat:type (stat filename)))))

(define (slash-suffix? name)
  (eq? #\/ (string-ref name (1- (string-length name)))))

(define (canonize-filename name)
  (if (directory? name)
      (if (slash-suffix? name)
          name
          (format #f "~a/" name))
      name))

(define (my-dirname name)
  (if (directory? name)
      (canonize-filename name)
      (dirname name)))
```

```
(define (file-name-completer string predicate all?)
    (if all?
        (all-completions string (files-in-parent-dir string) predicate)
        (try-completion string (files-in-parent-dir string) predicate)))

(define (dot-directory? name)
  (or ;(string=? "." name)
      ;(string=? ".." name)
      (string-suffix? "./" name)
      (string-suffix? "../" name)))

(define no-dot-files (negate dot-directory?))

(define*-public
  (read-file-name prompt #:key
                  (dir #f)
                  (default-file-name #f)
                  (initial #f)
                  (predicate no-dot-files)
                  (history (what-command-am-i?)))
  (completing-read prompt
                   filename-completion-function
                   #:predicate predicate
                   #:history history))
```

111 ⟨*minibuffer:test* 99b⟩+≡

```
(chdir (format #f "~a/~a" (getenv "ABS_TOP_SRCDIR") "test/minibuffer-test-dir"))
(check (dirname "") => ".")
(check (my-dirname "") => ".")
(check (my-dirname "../now") => "..")
(check (my-dirname "bin") => "bin/")
(check (my-dirname "bin/") => "bin/")
(check (my-dirname "mini") => ".")
(check (directory? ".") => #t)
(check (directory? "..") => #t)
(check (canonize-filename ".") => "./")
;(check (files-in-dir ".") => '())
(for-each (lambda (file-name-completer)
            (check (file-name-completer "mini" (const #t) #t) => '("minibuffer-a" "minibuffer-b"))
            (check (file-name-completer "mini" (const #t) #f) => "minibuffer-")
            (check (file-name-completer "bi" (const #t) #f) => "bin/")
            (check (file-name-completer "bin" (const #t) #f) => "bin/")
            (check (file-name-completer "bin/" (const #t) #f) => "bin/")
            ;; Get rid of the dot files.
            (check (file-name-completer "bin" no-dot-files #f) => "bin/run-test")
            (check (file-name-completer "bin/" no-dot-files #f) => "bin/run-test")
            (check (file-name-completer "ex" (const #t) #f) => "exam/")
            (check (file-name-completer "em" (const #t) #f) => "empty-dir/"))
          (list file-name-completer
                #;(lambda (string predicate all?)
                  (if all?
                      (stream->list (readline-completer->stream filename-completion-function string))
                      (filename-completion-function string #f)))))
(check (dirname "bin/") => ".")
(check (basename "bin/") => "bin")
(check (basename "bin/f") => "f")
(check (dirname "bin/f") => "bin")
(check (my-dirname "bin/") => "bin/")
(check (files-in-parent-dir "bin/") => '("bin/../" "bin/./" "bin/run-test"))
(check (files-in-parent-dir "bin") => '( "bin/../" "bin/./" "bin/run-test"))

(check (files-in-dir "bin/") => '("." ".." "run-test"))
(check (files-in-dir "bin") => '( "." ".." "run-test"))
(check (file-name-completer "bin/" no-dot-files #t) => '("bin/run-test"))

;(check (files-in-dir "..") => '())

(chdir (format #f "~a/~a" (getenv "ABS_TOP_SRCDIR") "test/minibuffer-test-dir/empty-dir"))
(check (file-name-completer "../ex" (const #t) #f) => "../exam/")
```

6.2.4 Minibuffer History

112a ⟨*minibuffer:procedure* 98d⟩+≡

```
(define*-public (make-history #:optional (list '()) (index #f))
  (make-cursor-list list (or index (length list))))

(set! minibuffer-history (make-history))

(define-public (history-insert! history value)
  (cursor-right-insert! history value)
  #;(fluid-set! minibuffer-history
              (list-insert! (fluid-ref minibuffer-history)
                        index
                        value)))

(define-public (history-ref history)
  (if (cursor-right? history)
   (cursor-right-ref history)
   #f)
  #;(list-ref (fluid-ref minibuffer-history) index))

(define-public (history-set! history value)
      (cursor-right-set! history value)
      #;(if (cursor-right? history)
      (cursor-right-set! history value)
      (cursor-right-insert! history value))
  #;(let ((lst (fluid-ref minibuffer-history)))
    (list-set! lst index value)
    (fluid-set! minibuffer-history lst))
  value)
```

112b ⟨*minibuffer:state* 97b⟩+≡

```
(define-public minibuffer-history #f)
(define history-symbol-map (make-hash-table))
```

112c ⟨*util:procedure* 98b⟩+≡

```
(define-public (list-insert! lst k val)
  "Insert val into list such that (list-ref list k) => val."
  (receive (pre post) (split-at! lst k)
    (append! pre (list val) post)))
```

112d ⟨*minibuffer:test* 99b⟩+≡

```
(let ((h (make-history '("3" "2" "1"))))
  (cursor-left! h)
    (check ( history-ref h) => "1")
    (history-set! h "a")
    (check ( history-ref h) => "a")
    (cursor-left! h)
    (check ( history-ref h) => "2")
    (cursor-right! h)
    (cursor-right! h)
    (check (cursor-list->list h) => '("3" "2" "a")))
```

Some commands for manipulating the minibuffer history.

113 ⟨*minibuffer:command* 102b⟩+≡

```
(define-interactive (previous-history-element #:optional (n 1))
  (define (previous-history-element* n)
    (cond
    ((> n 0)
     (if (cursor-left? minibuffer-history)
         (begin
           (cursor-left! minibuffer-history)
           (previous-history-element* (1- n)))
         (begin (minibuffer-message " [Beginning of history; no preceding item]")
                #;(previous-history-element 0))))
    ((< n 0)
     (if (cursor-right? minibuffer-history 2)
         (begin
           (cursor-right! minibuffer-history)
           (previous-history-element* (1+ n)))
         (begin (minibuffer-message " [End of history; no default available]")
                #;(previous-history-element 0))))
    ((= n 0)
     #f)))
  ;;
  (history-set! minibuffer-history (minibuffer-contents))
  (previous-history-element* n)
(with-buffer minibuffer (goto-char (point-min)))
(delete-minibuffer-contents minibuffer)
(insert (history-ref minibuffer-history))
(format #t "minibuffer-history ~a~%" minibuffer-history)
(history-ref minibuffer-history)

  #;(let* ((i (fluid-ref minibuffer-history-index))
         (j (+ i n))
         (history (fluid-ref minibuffer-history)))
    (pretty-print history)
    (pretty-print i)
    (pretty-print j)
    (cond ((< j 0)
           )
          ((>= j (length history))
           (minibuffer-message " [Beginning of history; no preceding item]"))
          (else
           (history-set! i (minibuffer-contents))
           (with-buffer minibuffer
            (goto-char (point-min)))
           (delete-minibuffer-contents minibuffer)
           (insert (list-ref history j))
           (fluid-set! minibuffer-history-index j)))))

(define-interactive (next-history-element #:optional (n 1))
  (previous-history-element (- n)))
```

File Layout

114a ⟨*file:minibuffer.scm* 114a⟩≡

```
(define-module (emacsy minibuffer)
  #:use-module (ice-9 optargs)
  #:use-module (ice-9 receive)
  #:use-module (ice-9 regex)
  #:use-module (ice-9 session)
  #:use-module (ice-9 gap-buffer)
  #:use-module (ice-9 pretty-print)
  #:use-module (ice-9 readline)
  #:use-module (srfi srfi-26)
  #:use-module (srfi srfi-41) ;; streams
  #:use-module (system vm program)
  #:use-module (oop goops)
  #:use-module (string completion)
  #:use-module (cursor-list)
  #:use-module (emacsy util)
  #:use-module (emacsy self-doc)
  #:use-module (emacsy keymap)
  #:use-module (emacsy event)
  #:use-module (emacsy buffer)
  #:use-module (emacsy command)
  #:use-module (emacsy block)
  #:use-module (emacsy klecl)
  #:use-module (emacsy kbd-macro)
  #:use-module (emacsy agenda)
  #:use-module (emacsy coroutine))

⟨minibuffer:macro (never defined)⟩
⟨minibuffer:class 97a⟩
⟨minibuffer:state 97b⟩
⟨minibuffer:procedure 98d⟩
⟨minibuffer:command 102b⟩
⟨minibuffer:keymap 97c⟩
⟨minibuffer:process 98a⟩
```

Layout for tests.

114b ⟨*file:minibuffer-test.scm* 114b⟩≡

```
(use-modules (emacsy minibuffer)
             (emacsy event)
             (emacsy klecl)
             (oop goops)
             (check))

(use-private-modules (emacsy minibuffer))

(set! emacsy-interactive? #t)

⟨+ Test Preamble (never defined)⟩
⟨minibuffer:test 99b⟩
⟨+ Test Postscript (never defined)⟩
```

6.3 Core

Now we're going to put in place some core functionality that makes Emacsy an Emacs-like library.

We need a global keymap.

116a ⟨*core:state* 116a⟩≡
```
(define-public global-map (make-keymap))
(define-public special-event-map (make-keymap))
```

116b ⟨*core:procedure* 116b⟩≡
```
(define-public (current-minor-mode-maps)
  (list))

(define-public (current-active-maps)
  `(,(current-local-map) ,@(map mode-map (buffer-modes (current-buffer))) ,global-map))

(set! default-klecl-maps current-active-maps)
```

And here are the essential key bindings.

116c ⟨*core:keybinding* 116c⟩≡
```
(define-key global-map "M-:"         'eval-expression)
(define-key global-map "M-x"         'execute-extended-command)
(define-key global-map "C-g"         'keyboard-quit)
(define-key global-map "C-x C-c"     'quit-application)
(define-key global-map "C-u"         'universal-argument)

(define-key special-event-map "C-g" 'keyboard-quit)
```

6.3.1 eval-expression

There is one command that I consider fundamental for an Emacs-like program. Whenever I'm presented with a program that claims to be Emacs-like, I try this out `M-: (+ 1 2)`. If it doesn't work then it may have Emacs-like key bindings, but it's not Emacs-like. That command is `eval-expression`. Let's write it.

116d ⟨*core:command* 116d⟩≡
```
(define-interactive
  (eval-expression
   #:optional
   (expression (read-from-string
                (completing-read "Eval: " apropos-completion-function))))
  (let ((value (eval expression (interaction-environment))))
    (message "~a" value)
    value))
```

116e ⟨*core:test* 116e⟩≡
```
(set! emacsy-interactive? #f)
(check (eval-expression '(+ 1 2)) => 3)
(set! emacsy-interactive? #t)
```

`read-from-string` parses the string into an expression.

116f ⟨*util:procedure* 116f⟩≡
```
(define-public (read-from-string string)
  (call-with-input-string string (lambda (port) (read port))))
```

6.3.2 execute-extended-command

The second fundamental command is `execute-extended-command` invoked with `M-x`.

117a ⟨*core:command* 116d⟩+≡
```
(define-interactive (execute-extended-command #:optional (n 1))
  ;(display "HERE!\n")
  (let ((str (completing-read "M-x " (completer global-cmdset))))
    (call-interactively (module-ref (current-module) (string->symbol str)))))
```

117b ⟨*core:command* 116d⟩+≡
```
(define-interactive (quit-application)
  (set! emacsy-quit-application? #t)
  (wait))
```

117c ⟨*core:state* 116a⟩+≡
```
(define-public emacsy-quit-application? #f)
```

6.3.3 universal-argument

117d ⟨*core:state* 116a⟩+≡
```
;(define-parameter universal-argument-queue (make-q) "This holds the current universal argument value
(define universal-argument-queue (make-q))
```

117e ⟨*core:procedure* 116b⟩+≡
```
(define-public (universal-argument-ref)
  (if (q-empty? universal-argument-queue)
      1
      (q-front universal-argument-queue)))

(define-public (universal-argument-pop!)
  (if (q-empty? universal-argument-queue)
      1
      (q-pop! universal-argument-queue)))

(define-public (universal-argument-push! arg)
  (q-push! universal-argument-queue arg))
```

This `universal-argument` command is written using a different style than is typical for interative Emacs commands. Most Emacs commands are written with their state, keymaps, and ancillary procedures as public variables. This style has a benefit of allowing one to manipulate or extend some pieces; however, there are some benefits to having everything encapsulated in this command procedure. For instance, if the minibuffer were written in this style, one could invoke recursive minibuffers.

118 ⟨*core:command* 116d⟩+≡

```
(define-interactive (universal-argument)
  "Universal argument is used as numerical input for many functions."
  (let ((count 0)
        (ua-keymap (make-keymap))
        (prompt "C-u ")
        ;(acceptable-chars (char-set-adjoin char-set:digit #\-))
        (done? #f))
    (define (add-to-prompt string)
      (set! prompt (string-concatenate (list prompt string " "))))
    (define (process-arg number)
      (add-to-prompt (number->string number))
      (set! count (+ (* count 10) number)))
    (define (my-undefined-command key-sequence events)
      (if (= count 0)
          (universal-argument-push! 4)
          (universal-argument-push! count))
      (set! done? #t)
      (set! echo-area "") ;; Clear the echo area.
      (for-each emacsy-event-unread (reverse events)))
    (define-key ua-keymap "1" (lambda () (process-arg 1)))
    (define-key ua-keymap "2" (lambda () (process-arg 2)))
    (define-key ua-keymap "3" (lambda () (process-arg 3)))
    (define-key ua-keymap "4" (lambda () (process-arg 4)))
    (define-key ua-keymap "5" (lambda () (process-arg 5)))
    (define-key ua-keymap "6" (lambda () (process-arg 6)))
    (define-key ua-keymap "7" (lambda () (process-arg 7)))
    (define-key ua-keymap "8" (lambda () (process-arg 8)))
    (define-key ua-keymap "9" (lambda () (process-arg 9)))
    (define-key ua-keymap "0" (lambda () (process-arg 0)))
    (define-key ua-keymap "-" (lambda ()
                                (set! count (- count))
                                (add-to-prompt "-")))
    (define-key ua-keymap "C-u" (lambda ()
                                  (when (= count 0)
                                    (set! count 4))
                                  (set! count (* count 4))
                                  (add-to-prompt "C-u")))
    (while (not done?)
      (primitive-command-tick prompt
                              #:keymaps (list ua-keymap)
                              #:undefined-command my-undefined-command))
    #;(let loop ((input (read-key prompt)))
        ;; We only admit numbers and a dash
        (if (and (is-a? input <key-event>)
                 (null? (modifier-keys input))
                 (char-set-contains? acceptable-chars (command-char input)))
            (begin
```

```
    (cond
     ((char=? (command-char input) #\-)
      (add-to-prompt "-")
      (set! count (- count)))
     (else
      (process-arg (string->number (string (command-char input))))))
    (loop (read-key prompt)))
  (begin
    (if (= count 0)
        (universal-argument-push! 4)
        (universal-argument-push! count))
    (emacsy-event-unread input))))))
```

Keyboard Macro Keybindings

119a ⟨*core:keybinding* 116c⟩+≡

```
(define-key global-map "C-x (" 'kmacro-start-macro)
(define-key global-map "C-x )" 'kmacro-end-macro)
(define-key global-map "C-x e" 'kmacro-end-and-call-macro)
```

Buffer Manipulation Keybindings

119b ⟨*core:keybinding* 116c⟩+≡

```
(define-key global-map "C-o"   'other-buffer)
(define-key global-map "C-x k" 'kill-buffer)
(define-key global-map "C-x b" 'switch-to-buffer)
```

One problem with this is I'd like to give completing-read a list of objects that will be converted to strings, but I'd like to get the object out rather than the string. I want something like this:

119c ⟨*core:test* 116e⟩+≡

```
(check (let* ((symbols '(aa ab c d)))
   (let-values
       (((to-string from-string) (object-tracker symbol->string)))
     (map from-string (all-completions "a" (map to-string symbols))))) => '(aa ab))
```

So let's write it. (This functionality has been baked into completing-read with the `#:to-string` keyword argument.)

119d ⟨*util:procedure* 116f⟩+≡

```
;; object-tracker :: (a -> b) -> ((a -> b), (b -> a))
(define-public (object-tracker a->b)
  (define (swap-cons c)
    (cons (cdr c) (car c)))
  (let ((cache (make-hash-table)))
    (values
     (lambda (x)
       (let ((y (a->b x)))
         (if (hash-ref cache y)
             (emacsy-log-warning "object-tracker has a duplicate for pairs ~a ~a" (cons x y) (swap-cor
         (hash-set! cache y x)
         y))
     (lambda (y)
       (or (hash-ref cache y) y)))))
```

120a ⟨*core:procedure* 116b⟩+≡
```
(define-interactive
  (switch-to-buffer
   #:optional
   (buffer
    (let-values (((to-string from-string) (object-tracker buffer-name)))
      (from-string (completing-read "Buffer: "
                                    (map to-string (buffer-list))))))
   (buffer-class-arg #f))
  (define (coerce buffer)
    (if (is-a? buffer <string>)
        (or (find (lambda (b) (string=? buffer (buffer-name b))) (buffer-list))
            buffer)
        buffer))
  (set! buffer (coerce buffer))
  (if (is-a? buffer <buffer>)
   ((@@ (emacsy buffer) primitive-switch-to-buffer) buffer)
   (if (is-a? buffer <string>)
    ;; Create a new buffer
    (let* ((buffer-class (or buffer-class-arg ⟨Choose a buffer class. 120c⟩))
           (new-buffer ⟨Create a new buffer of a certain class. 120b⟩))
          (add-buffer! new-buffer)
          new-buffer)
    (begin (message "Buffer or buffer-name expected.")
           #f))))
```

We will need to register classes that will be available to the user to instantiate.

120b ⟨*Create a new buffer of a certain class.* 120b⟩≡
```
(make buffer-class #:name buffer)
```

120c ⟨*Choose a buffer class.* 120c⟩≡
```
(if (= (length buffer-classes) 1)
    ;; If we only have one buffer class, use that.
    (car buffer-classes)
    ;; Otherwise, let the user choose one.
    (let*-values
        (((to-string from-string) (object-tracker
                                   (compose symbol->string class-name))))
      (let ((a-buffer-class
             (from-string (completing-read "Buffer Class: "
                                           (map to-string buffer-classes)))))
        (if (is-a? a-buffer-class <class>)
            a-buffer-class
            (begin
              (message "No such class ~a." a-buffer-class)
              (throw 'invalid-class))))))
```

120d ⟨*core:state* 116a⟩+≡
```
(define-public buffer-classes (list <text-buffer>))
```

6.3.4 Messages Buffer

120e ⟨*core:state* 116a⟩+≡
```
(define-public messages
  (make <text-buffer> #:keymap (make-keymap) #:name "*Messages*"))
```

121a ⟨*core:process* 121a⟩≡
```
(add-buffer! messages)
```

121b ⟨*core:procedure* 116b⟩+≡
```
(define echo-area "")

(define-public (emacsy-echo-area)
  echo-area)

(define-public (current-message)
  echo-area)

(define (emacsy-message . args)
  (let ((string (apply format #f args)))
   (with-buffer messages
                (insert string)
                (insert "\n"))
   (set! echo-area string)
   (if emacsy-interactive?
       (wait)
       (display string))
   string))

;; There's probably a better way to do this.
(set! message emacsy-message)
```

When the minibuffer is entered, we want to clear the echo-area. Because the echo-area is defined in core, it seems best to deal with it in core rather than placing echo-area handling code in minibuffer.

121c ⟨*core:procedure* 116b⟩+≡
```
(define-public (clear-echo-area)
  (set! echo-area ""))
```

121d ⟨*core:process* 121a⟩+≡
```
(add-hook! (buffer-enter-hook minibuffer)
           (lambda () (clear-echo-area)))
```

We want to be able to load a scheme file.

121e ⟨*core:command* 116d⟩+≡
```
(define-interactive
  (load-file #:optional (filename (read-file-name "Filename: ")))
  (catch #t
    #.\ (begin (load filename)
               (message "Loaded ~a." filename)
               #t)
    (lambda (key . args)
      (let ((error-msg
             (call-with-output-string
              #.\ (apply display-error #f % args))))
        (message "Failed to load ~a: ~a" filename error-msg)
        #f))))
```

These are most of the C API calls.

122 ⟨*core:procedure* 116b⟩+≡

```
(define-public (emacsy-message-or-echo-area)
  (if emacsy-display-minibuffer?
      (buffer-string minibuffer)
      echo-area))

(define-method-public (emacsy-mode-line)
  (emacsy-mode-line (current-buffer)))

;; XXX this should be moved into the (emacsy buffer) module.
(define-method-public (emacsy-mode-line (buffer <buffer>))
  (format #f "-:**- ~a    (~{~a~^ ~})" (buffer-name buffer) (map mode-name (buffer-modes buffer))))

(define-public (emacsy-minibuffer-point)
  (if emacsy-display-minibuffer?
      (point minibuffer)
      -1))

(define-public (emacsy-run-hook hook . args)
  (catch #t
    (lambda ()
      (if debug-on-error?
          (call-with-error-handling
           (lambda ()
             (apply run-hook hook args)
             #t))
          (with-backtrace*
           (lambda ()
             (apply run-hook hook args)
             #t))))
    (lambda (key . args)
      (emacsy-log-error "Hook ~a threw error '~a with args ~a " hook key args)
;      (emacsy-log-error "Resetting hook ~a" hook)
;      (reset-hook! hook)
      #f)))

(define-public emacsy-terminate-hook (make-hook))

(define-public (emacsy-terminate)
  (run-hook emacsy-terminate-hook))

(define-public (emacsy-tick)
  (define (my-tick)
    (update-agenda))

  (if debug-on-error?
      (call-with-error-handling
       (lambda ()
         (my-tick)))
      (with-backtrace*
       (lambda ()
         (my-tick)))))
```

We need to be able to deal with exceptions gracefully where ever they may pop up.

123a ⟨*core:test* 116e⟩+≡
```
(define (good-hook)
  #t)
(define (bad-hook)
  (throw 'some-error))
(define my-hook (make-hook 0))

(check-throw (run-hook my-hook) => 'no-throw)
(check-throw (emacsy-run-hook my-hook) => 'no-throw)
(check (emacsy-run-hook my-hook) => #t)
(add-hook! my-hook good-hook)
(check-throw (emacsy-run-hook my-hook) => 'no-throw)
(add-hook! my-hook bad-hook)
(check-throw (run-hook my-hook) => 'some-error)
(check-throw (emacsy-run-hook my-hook) => 'no-throw)
(check (emacsy-run-hook my-hook) => #f)
```

We want to be able to define variables that are not redefined if a source file or module is reloaded, just like `define-once`.

6.3.5 Mouse Movement

Sometimes we may want to track the motion events generated by a mouse. We don't do this all the time because it seems unnecessarily taxing.

123b ⟨*core:state* 116a⟩+≡
```
(define-public emacsy-send-mouse-movement-events? #f)
```

123c ⟨*core:macro* 123c⟩≡
```
(define-syntax-public track-mouse
  (syntax-rules ()
    ((track-mouse e ...)
     (in-out-guard  ;; This is different from dynamic-wind.
       (lambda () (set! emacsy-send-mouse-movement-events? #t))
       (lambda () e ...)
       (lambda () (set! emacsy-send-mouse-movement-events? #f))))))
```
Uses `in-out` 95a.

124a ⟨*core:procedure* 116b⟩+≡

```
(define* (read-from-mouse #:optional (prompt #f))
  (define (my-read-event)
    (if (and (pair? this-command-event)
             (mouse-event? (car this-command-event)))
        (let ((event (car this-command-event)))
          (set! this-command-event (cdr this-command-event))
          event)
          ;; XXX Should this be read-key or read-event?
        (read-event prompt)))
  (let loop ((event (my-read-event)))
    (if (mouse-event? event)
        ;; Got an event.
        (position event)
        (let ((canceled? #f))
          ;; Put this event back in the queue.
          (emacsy-event event)
          (catch
            'quit-command
            (lambda () (primitive-command-tick))
            (lambda (key . args)
              (emacsy-log-debug "READ-FROM-MOUSE CANCELED\n")
              (set! canceled? #t)))
          (if canceled?
              (throw 'quit-command 'quit-read-from-mouse)
              (loop (my-read-event)))))))
```

124b ⟨*core:test* 116e⟩+≡

```
(emacsy-discard-input!)
;(emacsy-key-event #\a)
(define mouse-event #f)
(agenda-schedule (colambda ()
                           (format #t "START~%")
                           (set! mouse-event (read-from-mouse))
                           (format #t "END~%")))
;(with-blockable )
;(block-tick)
;(check mouse-event => #f)
;(update-agenda)
(emacsy-mouse-event #(0 0) 1 'down)
(update-agenda)
(check-true mouse-event)
;(block-tick)
```

6.3.6 Command Loop

If we ever run out of command loops due to errors, we start a new one.

125a ⟨*core:procedure* 116b⟩+≡
```
(codefine (restart-command-loop)
  ;; Start another command-loop
          (while #t
            (emacsy-log-warning "NO COMMAND LOOPS; STARTING ANOTHER.")
            (command-loop)))

(codefine (non-interactive-command-loop)
  ;; Start another command-loop
          (emacsy-log-warning "STARTING NON-INTERACTIVE COMMAND LOOP.")
          (catch #t
            (lambda () (primitive-command-loop))
            (lambda args
              (format #t "stopping non-interactive command loop ~a" args)))
          (exit 0)
          #;(quit-application))

;(agenda-schedule restart-command-loop)

(define-public (emacsy-initialize interactive?)
  #;(when interactive?
    (agenda-schedule restart-command-loop))
  (agenda-schedule (if interactive?
                       restart-command-loop
                       non-interactive-command-loop))
  (set! emacsy-interactive? interactive?))
```

And if we have warning, we emit then through `emacsy-log-warning`.

125b ⟨*util:procedure* 116f⟩+≡
```
(define-public (emacsy-log-warning format-msg . args)
  (apply format (current-error-port) format-msg args)
  (newline (current-error-port)))
```

125c ⟨*core:process* 121a⟩+≡
```
(add-hook! no-blocking-continuations-hook restart-command-loop)
```

File Layout

126a ⟨*file:core.scm* 126a⟩≡

```
(define-module (emacsy core)
  #:use-module (ice-9 q)
  #:use-module (ice-9 optargs)
  #:use-module (ice-9 readline)
  #:use-module (oop goops)
  #:use-module (rnrs io ports)
  #:use-module (debugging assert)
  #:use-module (system repl error-handling)
  #:use-module (srfi srfi-1)  ;; take
  #:use-module (srfi srfi-11) ;; let-values
  #:use-module (srfi srfi-26) ;; cut cute
  #:use-module (convenience-lambda)
  #:use-module (emacsy util)
  #:use-module (emacsy self-doc)
  #:use-module (emacsy keymap)
  #:use-module (emacsy event)
  #:use-module (emacsy mode)
  #:use-module (emacsy buffer)
  #:use-module (emacsy command)
  #:use-module (emacsy block)
  #:use-module (emacsy klecl)
  #:use-module (emacsy kbd-macro)
  #:use-module (emacsy minibuffer)
  #:use-module (emacsy coroutine)
  #:use-module (emacsy agenda)
  #:replace (switch-to-buffer)
  #:export (read-from-mouse))
⟨core:macro 123c⟩
⟨core:class (never defined)⟩
⟨core:state 116a⟩
⟨core:procedure 116b⟩
⟨core:command 116d⟩
⟨core:keybinding 116c⟩
⟨core:process 121a⟩
```

Layout for tests.

126b ⟨*file:core-test.scm* 126b⟩≡

```
(use-modules (emacsy core)
             (emacsy event)
             (emacsy klecl)
             (oop goops)
             (srfi srfi-11))

(use-private-modules (emacsy core))

(set! emacsy-interactive? #t)
⟨+ Test Preamble (never defined)⟩
⟨core:test 116e⟩
⟨+ Test Postscript (never defined)⟩
```

6.4 Emacsy Facade

So that users of our library don't have to import all of our nicely partitioned modules individually, we'll expose a facade module that re-exports all of the public interfaces for each module.

127a ⟨*file:emacsy.scm* 127a⟩≡

```
(define-module (emacsy emacsy)
  #:use-module (convenience-lambda)
  #:use-module (emacsy util)
  #:use-module (emacsy self-doc)
  #:use-module (emacsy event)
  #:use-module (emacsy keymap)
  #:use-module (emacsy coroutine)
  #:use-module (emacsy agenda)
  #:use-module (emacsy command)
  #:use-module (emacsy mode)
  #:use-module (emacsy buffer)
  #:use-module (emacsy block)
  #:use-module (emacsy klecl)
  #:use-module (emacsy kbd-macro)
  #:use-module (emacsy minibuffer)
  #:use-module (emacsy core)
  #:use-module (emacsy help))
⟨emacsy:procedure 127b⟩
⟨emacsy:process 127c⟩
```

127b ⟨*emacsy:procedure* 127b⟩≡

```
(define (re-export-modules . modules)
  (define (re-export-module module)
    (module-for-each
     (lambda (sym var)
       ;;(format #t "re-exporting ~a~%" sym)
       (module-re-export! (current-module) (list sym)))
     (resolve-interface module)))
  (for-each re-export-module modules))
```

127c ⟨*emacsy:process* 127c⟩≡

```
(re-export-modules
 '(emacsy util)
 '(emacsy self-doc)
 '(emacsy keymap)
 '(emacsy event)
 '(emacsy mode)
 '(emacsy buffer)
 '(emacsy coroutine)
 '(emacsy agenda)
 '(emacsy command)
 '(emacsy block)
 '(emacsy klecl)
 '(emacsy kbd-macro)
 '(emacsy minibuffer)
 '(emacsy core)
 '(emacsy help))
```

128a ⟨*file:emacsy-test.scm* 128a⟩≡
```
⟨+ Lisp File Header 128b⟩
⟨+ Test Preamble 137c⟩
(eval-when (compile load eval)
           (module-use! (current-module) (resolve-module '(emacsy))))
⟨Definitions (never defined)⟩
⟨Tests (never defined)⟩
⟨+ Test Postscript 137d⟩
```

The header for Lisp files shown below.

128b ⟨+ *Lisp File Header* 128b⟩≡
```
#|
filename

DO NOT EDIT - automatically generated from emacsy.w.

⟨+ Copyright 128c⟩
⟨+ License (never defined)⟩
|#
```

128c ⟨+ *Copyright* 128c⟩≡
```
Copyright (C) 2012 Shane Celis
```

Appendices

Part II

Appendix

Appendix A

Support Code

A.1 Utility Module

The `util` module is a grab bag of all sorts of miscellaneous functionality. Rather than defining it here in one place. I thought it'd be best to define each piece where it is actually introduced and used.

134a ⟨*util:macro* 134a⟩≡
```
(define-syntax define-syntax-public
  (syntax-rules ()
    ((define-syntax-public name . body)
     (begin
       (define-syntax name . body)
       (export-syntax name)))))
(export-syntax define-syntax-public)
```

134b ⟨*util:macro* 134a⟩+≡
```
(define-syntax-public string-case
  (syntax-rules (else)
    ((_ str (else e1 ...))
     (begin e1 ...))
    ((_ str (e1 e2 ...))
     (when (string=? str e1) e2 ...))
    ((_ str (e1 e2 ...) c1 ...)
     (if (string=? str e1)
         (begin e2 ...)
         (string-case str c1 ...)))))
```

134c ⟨*util:macro* 134a⟩+≡
```
(define-syntax-public define-class-public
  (syntax-rules ()
    ((define-class-public name . body)
     (begin
       (define-class name . body)
       (export name)
       ))))
```

134d ⟨*util:macro* 134a⟩+≡
```
(define-syntax-public define-method-public
  (syntax-rules ()
    ((define-method-public (name . args) . body)
     (begin
       (define-method (name . args) . body)
       (export name)
       ))))
```

134e ⟨*util:macro* 134a⟩+≡
```
(define-syntax define-generic-public
  (syntax-rules ()
    ((define-generic-public name)
     (begin
       (define-generic name)
       (export name)))))
(export define-generic-public)
```

134f ⟨*util:procedure* 134f⟩≡
```
(define-public pp pretty-print)
```

135a ⟨*file:util.scm* 135a⟩≡
```
(define-module (emacsy util)
  #:use-module (ice-9 optargs)
  #:use-module (oop goops)
  #:use-module (ice-9 pretty-print)
  #:use-module (ice-9 receive)
  #:use-module (srfi srfi-1)
  #:use-module (debugging assert)
  #:use-module (system repl error-handling)
  ;#:export-syntax (define-syntax-public)
)
⟨util:state (never defined)⟩
⟨util:macro 134a⟩
⟨util:procedure 134f⟩
```

135b ⟨*util:procedure* 134f⟩+≡
```
(define-public (repeat-func count func)
  (if (<= count 0)
      #f
      (begin
        (func)
        (repeat-func (1- count) func))))
```

135c ⟨*util:macro* 134a⟩+≡
```
(define-syntax-public repeat
  (syntax-rules ()
    ((repeat c e ...)
     (repeat-func c (lambda () e ...)))))
```

A.2 Unit Testing Support

We want to be able to easily write and aggregate unit tests. It's not important to our project per se. We just need the utility. Our association list (alist) `unit-test` will hold the symbol of the function and the procedure.

135d ⟨*check/harness.scm* 135d⟩≡
```
(define-module (check harness)
  #:use-module (check)
  #:export (run-tests
            run-tests-and-exit)
  #:export-syntax (define-test))
```

Set up the variables.

135e ⟨*check/harness.scm* 135d⟩+≡
```
(define unit-tests '())
(define test-errors '())
```

We can register any procedure to a test name.

135f ⟨*check/harness.scm* 135d⟩+≡
```
(define (register-test name func)
  "Register a procedure to a test name."
  (set! unit-tests (acons name func unit-tests)))
```

Typically, users will define and register their tests with this macro.

136a ⟨*check/harness.scm* 135d⟩+≡
```
(define-syntax define-test
  (syntax-rules ()
    ((define-test (name args ...) expr ...)
     (begin (define* (name args ...)
        expr ...)
     (register-test 'name name)))))
```

We need to run the tests.

136b ⟨*check/harness.scm* 135d⟩+≡
```
(define (run-tests)
  (catch 'first-error
    (lambda ()
      ⟨handle each test 136c⟩)
    (lambda args
      #f)))
```

136c ⟨*handle each test* 136c⟩≡
```
(for-each
 (lambda (elt)
   (format #t "TEST: ~a\n" (car elt))
   ;;(pretty-print elt)
   (catch #t
     (lambda ()
       (with-throw-handler
        #t
        (lambda ()
          (apply (cdr elt) '()))
        (lambda args
          (set! test-errors (cons (car elt) test-errors))
          (format #t "Error in test ~a: ~a" (car elt) args)
          (backtrace))))
     (lambda args
       (throw 'first-error)
       #f)))
 (reverse unit-tests))
```

136d ⟨*check/harness.scm* 135d⟩+≡
```
(define (run-tests-and-exit)
  (run-tests)
  (check-report)
  (if (> (length test-errors) 0)
      (format #t "~a ERROR in tests: ~a." (length test-errors) (reverse test-errors))
      (format #t "NO ERRORs in tests."))
  (exit (if (and (= (length test-errors) 0) (= 0 (length check:failed))) 0 1)))
```

136e ⟨*test functions* 136e⟩≡
```
(define unit-tests '())

(define (register-test name func)
  (set! unit-tests (acons name func unit-tests)))
```

The function register-test does the work, but we don't want to require the user to call it, so we'll define a macro that will automatically call it.

137a ⟨*test macro* 137a⟩≡
```
(define-syntax define-test
  (syntax-rules ()
    ((define-test (name args ...) expr ...)
     (begin (define* (name args ...)
        expr ...)
     (register-test 'name name)))))
```

Finally, now we just need a way to run all the unit tests.

137b ⟨*run tests* 137b⟩≡
```
(define test-errors '())
(define (run-tests)
  (catch 'first-error
    (lambda () (for-each (lambda (elt)
                           (display "TEST: ")
                           (pretty-print elt)
                 (catch #t
                   (lambda ()
                     (with-throw-handler #t
                                         (lambda ()
                                           (apply (cdr elt) '()))
                                         (lambda args
                                           (set! test-errors (cons (car elt) test-errors))
                                           (format #t "Error in test ~a: ~a" (car elt) args)

                                           (backtrace))))
                   (lambda args
                     ;(throw 'first-error)
                     #f
                     )))
               (reverse unit-tests)))
    (lambda args
      #f)))
```

Finally, let's provide this as our testing preamble.

137c ⟨+ *Test Preamble* 137c⟩≡
```
(use-modules (check))
(use-modules (ice-9 pretty-print))

⟨test functions 136e⟩
⟨test macro 137a⟩
⟨run tests 137b⟩
```

Let's run these tests at the end.

137d ⟨+ *Test Postscript* 137d⟩≡
```
(run-tests)
(check-report)
(if (> (length test-errors) 0)
    (format #t "~a ERROR in tests: ~a." (length test-errors) (reverse test-errors))
    (format #t "NO ERRORs in tests."))
(exit (if (and (= (length test-errors) 0) (= 0 (length check:failed))) 0 1))
```

A.3 Vector Math

138a ⟨*vector-math-2.scm* 138a⟩≡
⟨*+ Lisp File Header* 128b⟩
;⟨*Vector Module* (never defined)⟩
⟨*Vector Definitions* 138b⟩

1. vector-component-usage

   The component of **a** in the **b** direction.

$$\begin{aligned} \text{comp}_{\mathbf{b}}\,\mathbf{a} &= \mathbf{a}\cdot\hat{\mathbf{b}} \\ &= \frac{\mathbf{a}\cdot\mathbf{b}}{||\mathbf{b}||} \end{aligned}$$

138b ⟨*Vector Definitions* 138b⟩≡
```
(define (vector-component a b)
    ;(string-trim-both
    #" ⟨vector-component-usage (never defined)⟩ "#
    ;char-set:whitespace)
 (/ (vector-dot a b) (vector-norm b)))
```

   Tried to define vector-component-usage to "Scalar projection"

2. Vector projection

   The vector projection of **a** on **b**.

$$\begin{aligned} \text{proj}_{\mathbf{b}}\,\mathbf{a} &= a_1\hat{\mathbf{b}} \\ a_1 &= \text{comp}_{\mathbf{b}}\,\mathbf{a} \end{aligned}$$

138c ⟨*Vector Definitions* 138b⟩+≡
```
(define (vector-projection a b)
 (vector* (vector-component a b) (vector-normalize b)))
```

Appendix B

Indices

This is an index of all the filenames, code fragments, and identifiers for the code.

B.1 Index of Filenames

B.2 Index of Fragments

B.3 Index of User Specified Identifiers