Emacsy: An Extensible, Embedable, Emacs-like Macro System

Shane Celis
shane.celis@gmail.com

Contents

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`

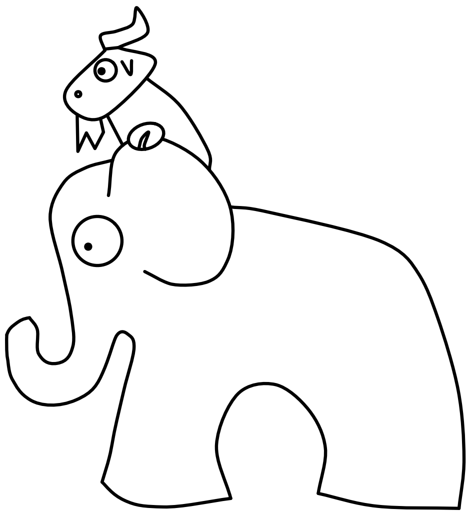

Figure 1.1: The proposed logo features a small gnu riding an elephant.

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

```
(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).

```
(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 embed legally

   Emacsy will be licensed under the LGPL.

3. 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.

4. 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.

5. Key bindings can be modified

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

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

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

7. Commands compose well

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

8. A small number of *interface* functions

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

9. 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

Usage

2.0.1 Minimal Emacsy Example

Part II
Implementation

Chapter 3

C API

Emacsy is divided into the following modules: klecl, buffer.

The minimal C API is given below.

⟨ *Prototypes ?* ⟩ ≡

```
/* Initialize Emacsy. */
int  emacsy_init(void);// XXX spell initialize out?

/* 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();

/* 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. */
void emacsy_terminate();
```

Fragment referenced in ?.

"emacsy.h" ?≡

```
/* file name

⟨ Copyright ? ⟩

⟨ License ? ⟩
*/

⟨ Begin Header Guard ? ⟩

⟨ Defines ?, ... ⟩

⟨ Prototypes ? ⟩

⟨ End Header Guard ? ⟩
```

Fragment uses f ?.

Here are the constants for the C API. TODO, add the EY_ prefix to these constants.

⟨ Defines ? ⟩ ≡

```
#define EY_MODKEY_COUNT     6

#define EY_MODKEY_ALT       1  // A
#define EY_MODKEY_CONTROL   2  // C
#define EY_MODKEY_HYPER     4  // H
#define EY_MODKEY_META      8  // M
#define EY_MODKEY_SUPER    16  // s
#define EY_MODKEY_SHIFT    32  // S

#define MOUSE_BUTTON_DOWN   0
#define MOUSE_BUTTON_UP     1
#define MOUSE_MOTION        2
```

Fragment defined by ?, ?.
Fragment referenced in ?.

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

⟨ Defines ? ⟩ ≡

```
#define EY_QUIT_APPLICATION   1
#define EY_ECHO_AREA_UPDATED  2
#define EY_MODELINE_UPDATED   4
```

Fragment defined by ?, ?.
Fragment referenced in ?.

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

⟨ Begin Header Guard ? ⟩ ≡

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

Fragment referenced in ?.

⟨ *End Header Guard ?* ⟩ ≡

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

Fragment referenced in ?.

The implementation of the API calls similarly named Scheme procedures.

3.1 emacsy_init

⟨ *Functions ?* ⟩ ≡

```
int emacsy_init()
{
  /* const char *load_path = "/Users/shane/School/uvm/CSYS-395-evolutionary-↩
    robotics/bullet-2.79/Demos/GuileDemo"; */
  /* scm_primitive_load_path(scm_from_locale_string(emacsy_file));  */
  scm_c_use_module("emacsy");

  /* load the emacsy modules */
  return 0;
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

3.2 emacsy_key_event

⟨ *Functions ?* ⟩ ≡

```
void emacsy_key_event(int char_code,
                      int modifier_key_flags)
{
  // XXX I shouldn't have to do a CONTROL key fix here.
  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-key-event"),
                    c,
                    modifier_key_flags_to_list(modifier_key_flags));
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

3.3 emacsy_mouse_event

⟨ *Functions* ? ⟩ ≡

```
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 MOUSE_BUTTON_UP:   state_sym = up_sym;     break;
  case MOUSE_BUTTON_DOWN: state_sym = down_sym;   break;
  case 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-mouse-event"),
                    scm_vector(scm_list_2(scm_from_int(x),
                                          scm_from_int(y))),
                    scm_from_int(button),
                    state_sym);
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

3.4 emacsy_tick

⟨ *Functions* ? ⟩ ≡

```
int emacsy_tick()
{
  int flags;
  (void) scm_call_0(scm_c_public_ref("emacsy",
                                     "emacsy-tick"));
  if (scm_is_true(scm_c_public_ref("emacsy",
                                   "emacsy-quit-application?")))
    flags |= EY_QUIT_APPLICATION;
  return flags;
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

3.5 emacsy_message_or_echo_area

⟨ *Functions* ? ⟩ ≡

```
char *emacsy_message_or_echo_area()
{
  return scm_to_locale_string(
    scm_call_0(scm_c_public_ref("emacsy",
                                "emacsy-message-or-echo-area")));
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

3.6 emacsy_modeline

⟨ *Functions* ? ⟩ ≡

```
char *emacsy_mode_line()
{
  return scm_to_locale_string(
    scm_call_0(scm_c_public_ref("emacsy",
                                "emacsy-mode-line")));
}
```

Fragment defined by ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

"emacsy.c" ?≡

```
#include "emacsy.h"
#include <libguile.h>

⟨ Utility Functions ? ⟩

⟨ Functions ?, ... ⟩
```

File defined by ?, ?, ?.

"emacsy.c" ?≡

```
int  emacsy_run_hook_0(const char *hook_name)
{
  /* This should be protected from all sorts of errors that the hooks
     could throw. */
  scm_run_hook(scm_c_private_ref("guile-user", hook_name),
               SCM_EOL);
  return 0;
}
```

File defined by ?, ?, ?.

"emacsy.c" ?≡

```
int  emacsy_minibuffer_point()
{
  return scm_to_int(
    scm_call_0(scm_c_public_ref("emacsy",
                                "emacsy-minibuffer-point")));
}
```

File defined by ?, ?, ?.

⟨ *Utility Functions* ? ⟩ ≡

```
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 < EY_MODKEY_COUNT; i++) {
    if (modifier_key_flags & (1 << i)) {
      list = scm_cons(scm_c_string_to_symbol(modifiers[i]), list);
    }
  }

  return list;
}
```

Fragment referenced in ?.

Chapter 4

KLECL

I expounded on the virtues of the Key Lookup Execute Command Loop (KLECL) in **??**. Now we're going to implement a KLECL.

"emacsy/klecl.scm" ?≡

```
⟨ Lisp File Header ? ⟩
(define-module (emacsy klecl)
  ⟨ Include Modules ? ⟩
  #:export ( ⟨ Exported Symbols ? ⟩ )
  #:export-syntax ( ⟨ Exported Syntax ? ⟩ ) )
⟨ Variables ? ⟩
⟨ Procedures ? ⟩
```

We will use the module check for most of our unit test needs.

"emacsy-tests.scm" ?≡

```

⟨ Lisp File Header ? ⟩
⟨ Test Preamble ? ⟩
(eval-when (compile load eval)
           (module-use! (current-module) (resolve-module '(emacsy))))
⟨ Definitions ? ⟩
⟨ Tests ? ⟩
⟨ Test Postscript ? ⟩
```

The header for Lisp files shown below.

⟨ Lisp File Header ? ⟩ ≡

```
#|
file name

DO NOT EDIT - automatically generated from emacsy.w.

⟨ Copyright ? ⟩
⟨ License ? ⟩
|#
```

Fragment referenced in ?, ?, ?, ?, ?.
Fragment uses `f` ?.

⟨ *Copyright* ? ⟩ ≡

```
Copyright (C) 2012 Shane Celis
```

Fragment referenced in ?, ?.

⟨ *Definitions* ? ⟩ ≡

```
(define (f x)
  (1+ x))
```

Fragment referenced in ?.
Fragment defines `f` ?, ?, ?, ?, ?, ?, ?.

⟨ *Tests* ? ⟩ ≡

```
(define-test (test-f)
  (check (f 1) => 2)
  (check (f 0) => 1)
)
```

Fragment referenced in ?.
Fragment defines `test-f` (never used).
Fragment uses `define-test` ?, ?, `f` ?.

Chapter 5

Optional Windows

Emacsy aims to offer the minimal amount of intrusion to acquire big gains in program functionality. Windows is an optional module for Emacsy. If you want to offer windows that behave like Emacs windows, you can, but you aren't required to.

"emacsy/windows.scm" ?≡

```
⟨ Lisp File Header ? ⟩
⟨ Module ? ⟩
⟨ Classes ?, ...  ⟩
⟨ State ?, ...  ⟩
⟨ Procedures ?, ...  ⟩
⟨ Commands ?, ...  ⟩
⟨ Key bindings ? ⟩
```

⟨ Module ? ⟩ ≡

```
(define-module (emacsy windows)
  #:use-module (oop goops)
  #:use-module (emacsy)
  ⟨ Include Modules ?, ...  ⟩
  #:export ( ⟨ Exported Symbols ?, ...  ⟩ )
)
```

Fragment referenced in ?.

5.1 Classes

The window class contains a renderable window that is associated with a buffer.

⟨ *Classes* ? ⟩ ≡

```
(define-class <window> ()
  (window-buffer #:accessor window-buffer #:init-value #f)
  (window-parent #:accessor window-parent #:init-value #f)
  (window-dedicated? #:accessor window-dedicated? #:init-value #f)
  (user-data #:accessor user-data #:init-value #f)
  (to-parent-transform #:accessor to-parent-transform #:init-value (↩
    make-identity-matrix 3))
  (from-parent-transform #:accessor from-parent-transform #:init-value (↩
    make-identity-matrix 3))
  )
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment defines `<window>` ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.

The internal window class contains other windows.

⟨ *Classes* ? ⟩ ≡

```
(define-class <internal-window> ()
  (window-children #:accessor window-children #:init-keyword #:window-children #:↩
    init-value '()) ; just two!
  (window-parent #:accessor window-parent #:init-value #f)
  (orientation #:accessor orientation #:init-keyword #:orientation #:init-value '↩
    vertical) ; or 'horizontal
  (size #:accessor size #:init-keyword #:size #:init-value .5)
  )
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment defines `<internal-window>` ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.

⟨ *Exported Symbols* ? ⟩ ≡

```
<window> <internal-window> <pixel-window>
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?, `<pixel-window>` ?, `<window>` ?.

⟨ *Classes* ? ⟩ ≡

```
(define-class <pixel-window> (<internal-window>)
  (window-child #:accessor window-child #:init-value #f)
  (pixel-size #:accessor pixel-size #:init-keyword #:pixel-size #:init-value '(640↩
     480)))
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment defines `<pixel-window>` ?, ?, ?, ?, ?, ?, ?.
Fragment uses `<internal-window>` ?.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (initialize (obj <pixel-window>) initargs)
  (next-method)
  (let ((child-window (make <window>)))
    (set! (window-parent child-window) obj)
    (set! (window-child obj) child-window)
  )
)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<pixel-window>` ?, `<window>` ?.

5.2 Procedures

⟨ *Procedures* ? ⟩ ≡

```
(define (window? o)
  (or (is-a? o <window>) (is-a? o <internal-window>) (is-a? o <pixel-window>))
  )
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?, `<pixel-window>` ?, `<window>` ?.

⟨ *Exported Symbols* ? ⟩ ≡

```
window?
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

⟨ *Tests* ? ⟩ ≡

```
  (check (window? root-window) => #t)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Procedures* ? ⟩ ≡

```
(define (window-live? o)
  (is-a? o <window>))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?.

⟨ *Tests* ? ⟩ ≡

```
  (check (window-live? root-window) => #t)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Procedures* ? ⟩ ≡

```
(define (frame-root-window)
  root-window)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *State* ? ⟩ ≡

```
(define root-window (make <window>))
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?.

⟨ *Include Modules* ? ⟩ ≡

```
  #:use-module (ice-9 match)
```

Fragment defined by ?, ?.
Fragment referenced in ?.

Emacs uses the edges of windows `(left top right bottom)`, but I'm more comfortable using bounded coordinate systems `(left bottom width height)`. So let's write some converters.

⟨ *Procedures* ? ⟩ ≡

```
(define (edges->bcoords edges)
  (match edges
   ((left top right bottom)
    (list left bottom (- right left) (- top bottom)))))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Tests* ? ⟩ ≡

```
(check (edges->bcoords '(0 1 1 0)) => '(0 0 1 1))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Procedures* ? ⟩ ≡

```
(define (bcoords->edges coords)
  (match coords
  ((x y w h)
    (list x (+ y h) (+ x w) y))))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Tests* ? ⟩ ≡

```
(check (bcoords->edges '(0 0 1 1)) => '(0 1 1 0))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

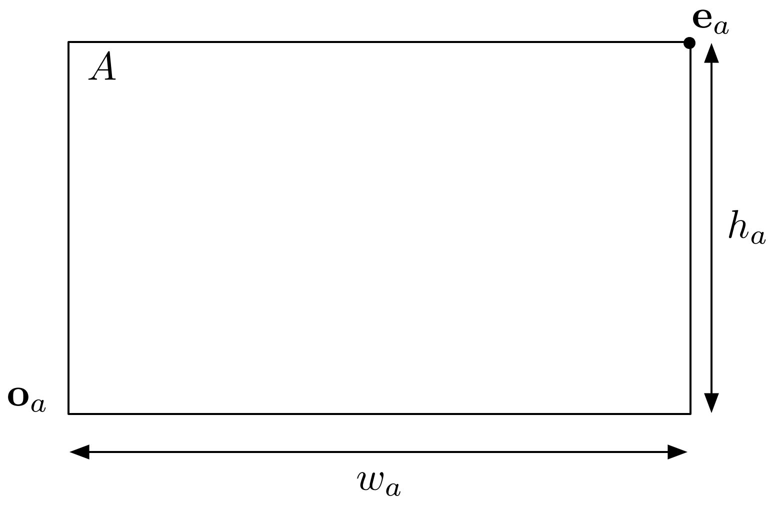

Figure 5.1: Window A can be fully described by two vectors: its origin $\mathbf{o}_a = (ox, oy)$ and its end $\mathbf{e}_a = (w_a, h_a)$.

The best way I can think to tile and scale all these windows is like this. Let's use a normalized bounded coordinates for the internal windows. This way the frame size can change and the pixel edges can be recomputed.

Imagine the frame has a width W and a height H. My root window has the bounded coordinates `(0 0 1 1)`. When I call `window-pixel-coords` on it, it will return `(0 0 W H)`.

Consider the case where my root window is split vertically in half. My root window would be an internal window with the same bounded coordinates as before. The top child, however, will have its pixel bounded coordinates as `(0 (/ H 2) W (/ H 2)`. And the bottom child will have `(0 0 W (/ H 2))`.

One way to think of this is every `<window>` takes up all its space; intrinsically, they are all set to `(0 0 1 1)`. The trick is each `<internal-window>` divides up the space recursively. So the internal window in the preceding example that was split vertically, it passes `0 .5 1 .5` to the top child and `0 0 1 .5`.

One thing we need is the absolute pixel bounded coordinates of the frame. Emacsy integrators need to set and update this appropriately.

⟨ *State ?* ⟩ ≡

```
(define emacsy-root-frame-absolute-pixel-bcoords '(0 0 640 320))
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

5.3 Units

There are two different units: pixel (px) and proportion (unitless but denoted pr for explicitness). The size of a pixel is the same for all windows reference frames. It is an absolute measure. However, the size of a proportion is relative to the window it is in. The end of every window $\mathbf{e}$ is $(1, 1)$ pr, or equivalently (w, h) px, and the origin of every window is $(0, 0)$ in pixels or proportions.

When the root window, or frame in Emacs parlance, is resized, we want each windows by default to resize proportionately. The windows will be tiled; therefore, it seems appropriate to use the unit of proportions as our representation over pixels. There will be some windows that will have a size of a particular pixel size, like the minibuffer window. A little bit of specialization to maintain a particular pixel height will require some callbacks or hooks.

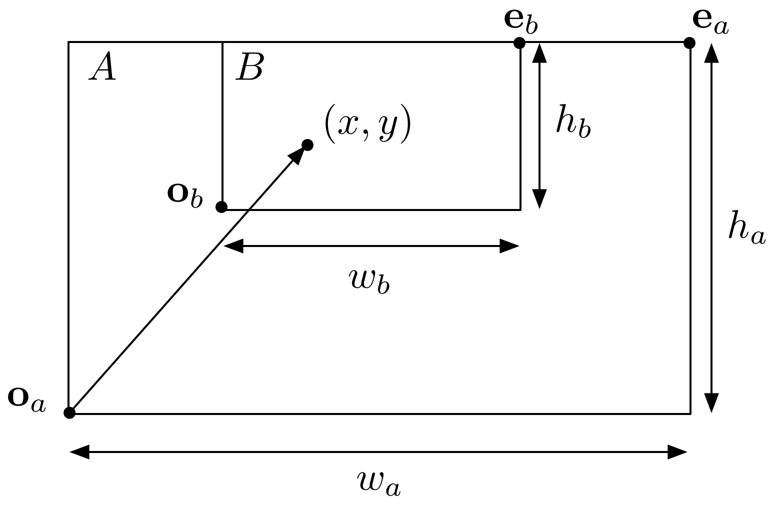

Figure 5.2: This diagram shows two windows A and B. Window B can be said to be a child of window A. Each window has its origin $\mathbf{o}$ and end $\mathbf{e}$ with its width w and height h. The point (x, y) may be referenced by either coordinate system denoted as follows. $(x, y)_A$ denotes the coordinates with respect to the A reference frame; $(x, y)_B$ denotes the coordinates with respect to the B reference frame.

5.4 Converting Between Window Reference Frames

Figure **??** shows a diagram of two windows: one parent and one child. The valid range of each variable is given below.

$$\mathbf{o}_b = (ox, oy) \tag{5.1}$$
$$ox \in [0, 1] \tag{5.2}$$
$$oy \in [0, 1] \tag{5.3}$$
$$w_b \in [0, 1 - ox] \tag{5.4}$$
$$h_b \in [0, 1 - oy] \tag{5.5}$$

Let's assume that we know the coordinates of the point with respect to RF B which we'll denote as $(x, y)_B$. How do we determine the coordinate wrt RF A, $(x', y')_A$? First, let's define some matrix operators: translate $\mathbf{T}(tx, ty)$ and scale $\mathbf{S}(sw, sh)$.

$$\mathbf{T}(tx, ty) = \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix} \tag{5.6}$$
$$\mathbf{S}(sw, sh) = \begin{bmatrix} sw & 0 & 0 \\ 0 & sh & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.7}$$

Now we can determine $(x', y')_A$.

$$(x', y')_A = \mathbf{T}(\mathbf{o}_b)\,\mathbf{S}(\mathbf{e}_b)\,(x, y)_B \tag{5.8}$$
$$\text{where } \mathbf{e}_b = (w_b, h_b) \tag{5.9}$$

Note that the multiplication between a 3×3 matrix and a two dimensional vector actually is shorthand for this

$$\mathbf{M}\,(x, y) \equiv \mathbf{M}\begin{bmatrix} x \\ y \\ 1 \end{bmatrix}.$$

This is a homogenous coordinate system that allows us to capture affine transformations like translation.

Let's denote the transformation to RF A from RF B as ${}_A\mathbf{M}_B$.

$$(x', y')_A = {}_A\mathbf{M}_B\,(x, y)_B \tag{5.10}$$

$$\text{where } {}_A\mathbf{M}_B = \mathbf{T}(\mathbf{o}_b)\,\mathbf{S}(\mathbf{e}_b) \tag{5.11}$$

And its inverse defines the opposite operation going to RF B from RF A.

$${}_A\mathbf{M}_B^{-1} = (\mathbf{T}(\mathbf{o}_b)\,\mathbf{S}(\mathbf{e}_b))^{-1} \tag{5.12}$$

$${}_A\mathbf{M}_B^{-1} = \mathbf{S}(\mathbf{e}_b)^{-1}\,\mathbf{T}(\mathbf{o}_b)^{-1} \tag{5.13}$$

$${}_A\mathbf{M}_B^{-1} = {}_B\mathbf{M}_A \tag{5.14}$$

5.4.1 Identities

Here are a few identities.

$$[\mathbf{o}_b]_B = (0, 0) \tag{5.15}$$

$$[\mathbf{e}_b]_B = (1, 1) \tag{5.16}$$

$$[\mathbf{o}_b]_A = {}_A\mathbf{M}_B\,(0, 0) \tag{5.17}$$

$$[\mathbf{e}_b]_A = {}_A\mathbf{M}_B\,(1, 1) \tag{5.18}$$

Again but in code this time.

⟨*Procedures* ?⟩ ≡

```
(define (translate-2d tx ty)
 (vector
  (vector 1 0 tx)
  (vector 0 1 ty)
  (vector 0 0 1)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment defines `translate-2d` ?, ?.

⟨*Procedures* ?⟩ ≡

```
(define (scale-2d sx sy)
 (vector
  (vector sx 0  0)
  (vector 0  sy 0)
  (vector 0  0  1)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment defines `scale-2d` ?, ?, ?, ?.

5.4.2 Split Window

Be careful with `deep-clone`. If you deep clone one window that has references to other windows, you will clone entire object graph.

⟨ *Commands* ? ⟩ ≡

```
(define-interactive (split-window #:optional
                     (window (selected-window))
                     (size 0.5)
                     (side 'below))
  (let* ((original-parent (window-parent window))
         (new-child (shallow-clone window))
         (internal-window (make <internal-window>
                           #:window-children (cons window new-child)
                           #:size size
                           #:orientation (if (memq side '(below above))
                                             'vertical
                                             'horizontal))))
    (set! (window-parent internal-window) original-parent)
    (set! (window-parent window)    internal-window)
    (set! (window-parent new-child) internal-window)
    (update-window internal-window)
  internal-window))
```

Fragment defined by ?, ?.
Fragment referenced in ?.
Fragment defines `split-window` ?, ?, ?, ?, ?, ?, ?, ?.
Fragment uses `<internal-window>` ?.

⟨ *Exported Symbols* ? ⟩ ≡

```
 split-window
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.
Fragment uses `split-window` ?.

⟨ *Procedures* ? ⟩ ≡

```
(define (selected-window)
  current-window)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *State* ? ⟩ ≡

```
(define current-window #f)
```

Fragment defined by ?, ?, ?.
Fragment referenced in ?.

If the internal window size is changed, we want to update the sizes of its children. Also, normally we'd only need to keep one matrix and just invert it as necessary; however, I haven't written a matrix solver routine, so I'm just going to construct the matrix and its inverse. (I wish guile-num were around.)

⟨ *Procedures ?* ⟩ ≡

```
(define-method (update-window (window <internal-window>))
 (let ((children (window-children window)))
  (if (eq? (orientation window) 'vertical)
    ⟨ Update vertical window. ? ⟩
    ⟨ Update horizontal window. ? ⟩)))
```

Fragment defined by ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?.

⟨ *Update vertical window. ?* ⟩ ≡

```
(let ((top-size (size window))
      (bottom-size (- 1 (size window))))
 (let ((top (car children))
       (to-parent (matrix. (translate-2d 0. top-size) (scale-2d 1. top-size)))
       (from-parent (matrix. (scale-2d 1. (/ 1 top-size)) (translate-2d 0. (- ↩
   top-size)))))
  (set! (to-parent-transform top) to-parent)
  (set! (from-parent-transform top) from-parent))
 (let ((bottom (cdr children))
       (to-parent (matrix. (translate-2d 0. 0.) (scale-2d 1. bottom-size)))
       (from-parent (matrix. (scale-2d 1. (/ 1 bottom-size)) (translate-2d 0. 0.))↩
   ))
  (set! (to-parent-transform bottom) to-parent)
  (set! (from-parent-transform bottom) from-parent)))
```

Fragment referenced in ?.
Fragment uses `scale-2d` ?, `translate-2d` ?.

⟨ *Update horizontal window. ?* ⟩ ≡

```
(let ((left-size (size window))
        (right-size (- 1 (size window))))
  (let ((left (car children))
        (to-parent (matrix. (translate-2d 0. 0.) (scale-2d left-size 1.)))
        (from-parent (matrix. (scale-2d (/ 1 left-size) 1.) (translate-2d 0. 0.)))↩
   )
   (set! (to-parent-transform left) to-parent)
   (set! (from-parent-transform left) from-parent))
  (let ((right (cdr children))
        (to-parent (matrix. (translate-2d left-size 0.) (scale-2d right-size 1.)))
        (from-parent (matrix. (scale-2d (/ 1 right-size) 1.) (translate-2d (- ↩
   left-size) 0.))))
   (set! (to-parent-transform right) to-parent)
   (set! (from-parent-transform right) from-parent)))
```

Fragment referenced in ?.
Fragment uses `scale-2d` ?, `translate-2d` ?.

⟨ *Include Modules ?* ⟩ ≡

```
#:use-module (vector-math)
```

Fragment defined by ?, ?.
Fragment referenced in ?.

5.4.3 Window Project

Let's project a point in the current window to the point in its ultimate parent window.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-project (window <window>) position)
  (let ((parent-position (matrix. (to-parent-transform window) position)))
    (if (window-parent window)
      (window-project (window-parent window) parent-position)
      parent-position)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?.

For internal-windows, we just pass the information through.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-project (window <internal-window>) position)
  (if (window-parent window)
    (window-project (window-parent window) position)
    position))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-project (window <pixel-window>) position)
  (let ((psize (pixel-size window)))
    (matrix. (scale-2d (car psize) (cadr psize)) position)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<pixel-window>` ?, `scale-2d` ?.

⟨ *Projection Tests* ? ⟩ ≡

```
(check (window-project window #(0 0 1)) => #(0. .5 1.))
(check (window-project window #(1. 1. 1.)) => #(1. 1. 1.))
(check (window-unproject window #(0 .5 1.)) => #(0. 0. 1.))
(check (window-unproject window #(1. 1. 1.)) => #(1. 1. 1.))
```

Fragment referenced in ?.

5.4.4 Window Unproject

Let's unproject from a point in the ultimate parent window to a point in the given window. Note that if the point is not within the bounds of the given window, the resulting point will not be within $[0, 1]^2$.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-unproject (window <window>) position)
  (let ((parent-position (window-unproject (window-parent window) position)))
    (if (window-parent window)
     (matrix. (from-parent-transform window) parent-position)
      parent-position)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?.

For internal-windows, we just pass the information through.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-unproject (window <internal-window>) position)
  (if (window-parent window)
    (window-unproject (window-parent window) position)
    position))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?.

⟨ *Procedures* ? ⟩ ≡

```
(define-method (window-unproject (window <pixel-window>) position)
  (let ((psize (pixel-size window)))
  (matrix. (scale-2d (/ 1 (car psize)) (/ 1 (cadr psize))) position)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<pixel-window>` ?, `scale-2d` ?.

⟨ *Procedures* ? ⟩ ≡

```
(define (window-pixel-bcoords window)
  (let ((origin (window-project window #(0 0 1)))
        (end    (window-project window #(1 1 0))))
    (list
     (vector-ref origin 0)
     (vector-ref origin 1)
     (vector-ref end 0)
     (vector-ref end 1))))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Tests* ? ⟩ ≡

```
(define i-window (make <internal-window>))
(define window (make <window>))
(check (window? i-window) => #t)
(check (window? window) => #t)
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?, `<window>` ?.

Let's test window splitting.

⟨ *Tests* ? ⟩ ≡

```
(check (procedure? split-window) => #t)
(define s-window (split-window window))
(check (is-a? s-window <internal-window>) => #t)
(check (window-pixel-bcoords s-window) => '(0 0 1 1))
(check (window-pixel-bcoords window) => '(0. .5 1. .5))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?, `split-window` ?.

Let's test window splitting with a different size value.

⟨ *Tests* ? ⟩ ≡

```
(define small-window (make <window>))
(define parent-window (split-window small-window 0.2))
(define big-window (cdr (window-children parent-window)))
(check (orientation parent-window) => 'vertical)
(check (window-pixel-bcoords small-window) => '(0. .2 1. .2))
(check (window-pixel-bcoords big-window) => '(0. 0. 1. .8))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?, `split-window` ?.

Let's test window splitting with a different orientation.

⟨ *Tests* ? ⟩ ≡

```
(define left-window (make <window>))
(define parent-window-2 (split-window left-window 0.2 'right))
(define right-window (cdr (window-children parent-window-2)))
(check (orientation parent-window-2) => 'horizontal)
(check (window-pixel-bcoords left-window) => '(0. 0. .2 1.))
(check (window-pixel-bcoords right-window) => '(.2 .0 .8 1.))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?, `split-window` ?.

Let's test the pixel-window at the top of the hierarchy.

⟨ *Tests ?* ⟩ ≡

```
(define pixel-window (make <pixel-window> #:pixel-size '(500 400)))

;(update-window pixel-window)
;(define sub-window   (window-child pixel-window))
(define sub-window   (make <window>))
(check (window? pixel-window) => #t)
(check (window? sub-window) => #t)
(set! (window-parent sub-window) pixel-window)
(set! (window-child pixel-window) sub-window)
(check (window-child pixel-window) => sub-window)
(check (window-project sub-window #(1. 1. 1.)) => #(500. 400. 1.))
(check (window-project sub-window #(0. 0. 1.)) => #(0. 0. 1.))
(format #t "Splitting the window\n")
(define sub-window-2 (split-window sub-window))
(check (window-project sub-window #(1. 1. 1.)) => #(500. 400. 1.))
(check (window-project sub-window #(0. 0. 1.)) => #(0. 200. 1.))

(check (window-unproject sub-window #(0. 200. 1.)) => #(0. 0. 1.))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<pixel-window>` ?, `<window>` ?, `split-window` ?.

5.4.5 Window List

⟨ *Procedures ?* ⟩ ≡

```
(define-method (window-tree (w <internal-window>))
  (let ((cs (window-children w)))
    (list (window-tree (car cs))
          (window-tree (cdr cs)))))

(define-method (window-tree (w <window>))
  w)

(define-method (window-tree (w <pixel-window>))
  (window-tree (window-child w)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<internal-window>` ?, `<pixel-window>` ?, `<window>` ?.

⟨ *Procedures ?* ⟩ ≡

```

(define (flatten x)
    (cond ((null? x) '())
          ((not (pair? x)) (list x))
          (else (append (flatten (car x))
                        (flatten (cdr x))))))

(define* (window-list #:optional (w root-window))
  (flatten (window-tree w)))
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.

⟨ *Tests* ? ⟩ ≡

```
(let* ((w (make <window>))
       (sw (split-window w))
       (c (cdr (window-children sw)))
       (sc (split-window c))
  )

  (check (window-list w) => (list w))
  (check (window-tree sw) => (list w c))
  (check (window-list sw) => (list w c))
  (check (window-list sw) => (list w c #f))
  )
```

Fragment defined by ?, ?, ?, ?, ?, ?, ?, ?, ?, ?.
Fragment referenced in ?.
Fragment uses `<window>` ?, `split-window` ?.

5.5 Window Commands

⟨ *Commands* ? ⟩ ≡

```
(define-interactive (split-window-below #:optional (size .5))
  (split-window (selected-window) size 'below))
```

Fragment defined by ?, ?.
Fragment referenced in ?.
Fragment uses `split-window` ?.

5.6 Window Key Bindings

It will come as no surprise that these key bindings will mimic the behavior of Emacs.

⟨ *Key bindings* ? ⟩ ≡

```
(define-key global-mode-map (kbd "C-x 2") 'split-window-below)
```

Fragment referenced in ?.
Fragment uses `split-window` ?.

`"windows-tests.scm"` ?≡

```
⟨ Lisp File Header ? ⟩
⟨ Test Preamble ? ⟩

(use-modules (emacsy) (emacsy windows))
(eval-when (compile load eval)
           (module-use! (current-module) (resolve-module '(emacsy windows))))
⟨ Tests ?, ... ⟩
⟨ Projection Tests ? ⟩

⟨ Test Postscript ? ⟩
```

Part III

Appendix

Appendix A

Support Code

A.1 Literate Programming Support

All the code for this project is generated from `emacsy.w`. To ease debugging, it is helpful to have the debug information point to the place it came from in `emacsy.w` and not whatever source file it came from. The program `nuweb` has a means of providing this information that works for C/C++ via the line pragma. Scheme does not support the line pragma, but the reader can fortunately be extended to support it.

An example use of it might look like this:

```
(define (f x)
#line 314 "increment-literately.w"
  (+ x 1))
```

BUG: The line pragma ends up littering the source with zero length strings, which often doesn't matter, but it can't be used everywhere especially within a particular form. I'm not entirely sure how to fix that.

⟨*Line Pragma Handler ?*⟩ ≡

```
(lambda (char port)
  (let* ((ine (read port))
         (lineno (read port))
         (filename (read port)))
    (if (not (eq? ine 'ine))
        (error (format #f "Expected '#line <line-number> <filename>'; got '#~a~a ~↩
    a \"~a\"'." char ine lineno filename)))
    (set-port-filename! port filename)
    (set-port-line! port lineno)
    (set-port-column! port 0)
    ""))
```

Fragment referenced in ?.
Fragment uses `f` ?.

One problem that popped up was I sometimes wanted to include pieces of documentation in embedded strings. Something that might end up looking like this in the source code:

```

(define (f x)
  "#line 352 "emacsy.w"
   The function f and its associated function h...
   #line 362 "emacsy.w"
  "
  ...
```

The above code will see a string "#line 352 " followed by a bare symbol emacsy.w, which will not do. To get around this, I implemented another reader extension that will strip out any #l lines within it.

⟨*Liberal String Quote Reader ?*⟩ ≡

```
(lambda (char port)
  (let ((accum '()))
    (let loop ((entry (read-char port)))
      (if (or (eof-object? entry)
              (and (char=? #\" entry)
                   (char=? #\# (peek-char port))
                   (begin (read-char port)
                          #t)))
          ;; We're done
          (apply string (reverse accum))
          (begin
            (if (and (char=? #\# entry)
                     (char=? #\l (peek-char port)))
                ;; Drop this line
                (begin (read-line port)
                       (loop (read-char port)))
                (begin
                  ;; Keep and loop
                  (set! accum (cons entry accum))
                  (loop (read-char port))))))))))
```

Fragment referenced in ?.

"my-line-pragma.scm" ?≡

```
(define-module (my-line-pragma)
  #:use-module (ice-9 rdelim))

(eval-when (compile load eval)
 (read-hash-extend #\l ⟨Line Pragma Handler ?⟩)
 (read-hash-extend #\" ⟨Liberal String Quote Reader ?⟩))
```

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-tests` will hold the symbol of the function and the procedure.

"check/harness.scm" ?≡

```
(define-module (check harness)
  #:use-module (check)
  #:export (run-tests
            run-tests-and-exit)
  #:export-syntax (define-test))
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment uses `define-test` ?, ?, `run-tests` ?.

Set up the variables.

"check/harness.scm" ?≡

```
(define unit-tests '())
(define test-errors '())
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment uses test-errors ?, unit-tests ?.

We can register any procedure to a test name.

"check/harness.scm" ?≡

```
(define (register-test name func)
  "Register a procedure to a test name."
  (set! unit-tests (acons name func unit-tests)))
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment uses register-test ?, unit-tests ?.

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

"check/harness.scm" ?≡

```
(define-syntax define-test
  (syntax-rules ()
    ((define-test (name args ...) expr ...)
     (begin (define* (name args ...)
        expr ...)
     (register-test 'name name)))))
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment defines define-test ?, ?, ?.
Fragment uses register-test ?.

We need to run the tests.

"check/harness.scm" ?≡

```
(define (run-tests)
  (catch 'first-error
    (lambda ()
      ⟨handle each test ?⟩)
    (lambda args
      #f)))
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment uses f ?, run-tests ?.

⟨*handle each test ?*⟩ ≡

```
(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))
```

Fragment referenced in ?.
Fragment uses `f` ?, `test-errors` ?, `unit-tests` ?.

`"check/harness.scm"` ?≡

```
(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)))
```

File defined by ?, ?, ?, ?, ?, ?.
Fragment uses `run-tests` ?, `test-errors` ?.

⟨*test functions ?*⟩ ≡

```
(define unit-tests '())

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

Fragment referenced in ?.
Fragment defines `register-test` ?, ?, ?, `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.

⟨*test macro* ?⟩ ≡

```
(define-syntax define-test
  (syntax-rules ()
    ((define-test (name args ...) expr ...)
     (begin (define* (name args ...)
        expr ...)
     (register-test 'name name)))))
```

Fragment referenced in ?.
Fragment defines `define-test` ?, ?, ?.
Fragment uses `register-test` ?.

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

⟨*run tests* ?⟩ ≡

```
(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)))
```

Fragment referenced in ?.
Fragment defines `run-tests` ?, ?, ?, ?, `test-errors` ?, ?, ?, ?.
Fragment uses `f` ?, `unit-tests` ?.

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

⟨*Test Preamble* ?⟩ ≡

```
(use-modules (check))
(use-modules (ice-9 pretty-print))

⟨test functions ?⟩
⟨test macro ?⟩
⟨run tests ?⟩
```

Fragment referenced in ?, ?.

Let's run these tests at the end.

⟨ *Test Postscript* ? ⟩ ≡

```
(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))
```

Fragment referenced in ?, ?.
Fragment uses `run-tests` ?, `test-errors` ?.

A.3 Vector Math

`"vector-math-2.scm"` ?≡

```
⟨ Lisp File Header ? ⟩
;⟨ Vector Module ? ⟩
⟨ Vector Definitions ?, ... ⟩
```

1. Scalar projection

   The component of **a** in the **b** direction.

$$\begin{aligned} \operatorname{comp}_{\mathbf{b}} \mathbf{a} &= \mathbf{a} \cdot \hat{\mathbf{b}} \\ &= \frac{\mathbf{a} \cdot \mathbf{b}}{||\mathbf{b}||} \end{aligned}$$

⟨ *Vector Definitions* ? ⟩ ≡

```
(define (vector-component a b)
    ;(string-trim-both
    #" ⟨ vector-component-usage ? ⟩ "#
    ;char-set:whitespace)
 (/ (vector-dot a b) (vector-norm b)))
```

Fragment defined by ?, ?.
Fragment referenced in ?.

⟨ *vector-component-usage* ? ⟩ ≡
    Scalar projection◇

Fragment referenced in ?.

2. Vector projection

   The vector projection of **a** on **b**.

$$\begin{aligned} \operatorname{proj}_{\mathbf{b}} \mathbf{a} &= a_1 \hat{\mathbf{b}} \\ a_1 &= \operatorname{comp}_{\mathbf{b}} \mathbf{a} \end{aligned}$$

⟨ *Vector Definitions* ? ⟩ ≡

```
(define (vector-projection a b)
 (vector* (vector-component a b) (vector-normalize b)))
```

Fragment defined by ?, ?.
Fragment referenced in ?.

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