GNU Emacs

Programming

Robert Krawitz

21 January 1988

This work was originally written for a Project Athena seminar presented in January 1986. It was rewritten, and new sections added, for a public seminar presented in January 1987 and for publication as a standalone document. It was further revised in January 1988.

Introduction

This handout accompanies IAP Activity # 1740, Programming GNU Emacs. This handout covers in detail all information presented in the activity, along with numerous examples.

This activity covers everything from simple customizations to writing major new code, along with hints for finding things. It assumes working knowledge of GNU Emacs along with knowledge of Lisp or Scheme at the level of 6.001.

Emacs is written in C, but it has an integral Lisp extension language, called *emacs-lisp* built in. The vast majority of the functions in emacs are written in this Lisp; only about 500 functions are hard coded in C.

1. Basics

There are certain things that you should know about the editor before actually sitting down and writing code. The Help system and the Info system are basic resources that you should be comfortable using.

1.1. The *Help* System

The GNU Emacs *help* command, which is normally bound to `C-h`, is the first place to look for information about some feature of GNU Emacs. Many of the options to *help* are used by people who simply want to edit text; we won't discuss these. However, several of these options are of more use to programmers; these we will discuss here.

Don't worry if you don't understand all of the terms used here; many of them refer to things we'll be discussing later.

1.2. Useful *Help* Options

- a This is bound by default to the function `command-apropos`, which finds all commands that match a particular regular expression (*regexp*). A *command* is a function that can be called from the keyboard, either by means of a sequence of keystrokes or by use of `M-x`. Not all functions are commands; we'll discuss the difference later.

  This command gives you a buffer of all matching commands, and one line of documentation on each command.

- `f` This gives the full documentation on any function, command or otherwise. You might, for example, use `C-ha` to find all commands matching a particular regexp, and then use `C-hf` to get the full documentation on the function. You can type part of the function name, and then use `spc`, `tab`, or `ret` to complete the name of the function.

- b This prints out a chart of all key bindings in use in the current buffer; i. e. what commands will be executed for any keystrokes that you might type. This is sometimes useful when modifying key bindings. This generates the well-known ‘‘wallchart’’.

- v This is similar to `C-hf`, except that it gives you documentation on a Lisp variable rather than on a function.

- `s` This describes the current syntax table. The syntax table is used by various functions to determine the ‘‘syntax’’ of various characters (e. g. whitespace, word constituent, parenthesis, etc.).
- `i` Run *info*. This puts you in the *info* system, which contains documentation on various topics. If you have used `info` or `xinfo` on ITS or TOPS-20, then you’ll know how to use this; if not, then read the tutorial, following instructions. It’s very simple to learn.
- `M-x apropos` Like `C-ha`, except that it searches for all functions and variables matching a given pattern. This is not by default bound to any key in *help*, but it is a very useful function for programmers. I find it convenient to bind it to `C-ha`.

1.3. Using *Help* Effectively

These functions are very powerful when used effectively. Apropos is an excellent way of finding what exists, but you have to be careful to select something reasonably pertinent to what you’re trying to do. Common things that programmers need to know about are "buffer", "list", "vector", "string", "search", etc. Examining the complete documentation on these functions is useful, as is finding the list code using tags, as described below.

Normally, things pertaining to a particular major mode include the name of the mode in their symbol names. For example, all *rmail* functions include the word ‘‘rmail’’ in them. This is a convention that should be used in all lisp code that you may write. Since gnuemacs doesn’t have packages to break up the name space, it is the programmer’s task to make the names meaningful.

1.4. Useful Major Modes

A *major mode* is a set of functions, variables, and key bindings that are defined in order to make editing specific kinds of files easier. We’ll be discussing these later on. For the time being, you should know about two major modes that you’ll use frequently when writing Lisp code.

- *Emacs-Lisp mode* is the preferred major mode for editing files of lisp code for emacs. It does standard lisp indenting, along with a special command `(C-M-x)` to evaluate the current top-level form that the cursor is over.
- *Lisp Interaction Mode* is an interactive lisp listener, such as that on the Lisp Machine, or the Scheme listener. RMS set it up as the startup buffer; we changed the startup buffer to a value controlled by a variable `startup-major-mode`, so you’ll want to *setq* this variable to `'lisp-interaction-mode`. This permits you to type lisp forms, and then evaluate them by typing `lf`. It is a very good way to examine the environment, and to play around with hairy lisp constructs.

2. Emacs Lisp Fundamentals

Emacs Lisp is similar to Common Lisp[1], but missing many features. Packages, structures, flavors,

[1] If you are not familiar with Lisp, you should find a book on Lisp, and learn the basics of the language. Two good books are Winston & Horn, and Steele. You might want to try Sussman & Abelson, but beware that their book uses Scheme, which is slightly different from Common Lisp.

readtables, and many other advanced features are missing, but these are not normally needed for editing-related tasks. Recursion is not as efficient as it is in most lisps, but iteration in the form of a `while` function is supported, and is normally more useful. Extensive text manipulation functions are provided instead.

Note that emacs-lisp is dynamically scoped; i. e., all variables are special. This is not compatible with Common Lisp, in which all variables are lexically scoped unless declared special. Thus, you can bind any variable with `let`. For example, if you are working in a read-only buffer and wish to make it read-write temporarily inside a function, all you need to do is execute whatever will write to the buffer inside a form

```
(let ((buffer-read-only nil))
  (insert "Now I can insert text"))
```

This is how rmail modifies its buffer, while leaving it read-only.

Emacs-lisp does have simplified versions of Common Lisp features such as signals and signal handling (not to be confused with Unix signals!) and unwind-protect; these are very useful in many situations, such as when you want to ensure that a buffer is in a consistent state

2.1. Definition of Lisp Functions

The vast majority of the code of GNU Emacs is contained in the Lisp code. A relatively few functions (572 in version 18.35) are defined in C code; these are either things that are drive low-level functionw or perform system-related tasks, or are things that would be extremely inefficient in higher-level code. Other than that, the vast majority of the code is written in Lisp and can be rewritten very easily. It is the Lisp-defined functions that we will discuss.

Keyboard macros are another mechanism that actually defines Lisp functions. We won't discuss these because they can only be used for relatively trivial tasks and are easy enough to learn on your own.

3. Special Features of Emacs Lisp

Emacs Lisp supports many features that support editing-related functions that are fundamental parts of the Lisp system. For example, buffers are Lisp objects that can be manipulated using the standard Lisp functions as well as many others that specifically operate on buffers. A wide variety of these features exist; we will discuss these individually.

3.1. Buffers

A *buffer* is an object that contains text. A buffer may be associated with a file, but it need not be. For example, the `*scratch*` buffer that is displayed when emacs starts is not associated with a specific file, but a buffer that is created when you visit a file is.

Each buffer has a *major mode* associated with it, such as text mode or `C mode` or `rmail mode`.

Modes will be covered in more detail later on. The system associates other state with each buffer, most of which is not interesting to the programmer.

In addition, buffers can have *local variables* associated with them, which means that the value of the variable changes when the current buffer is changed, by means of `set-buffer`, `switch-to-buffer`, `pop-to-buffer`, etc. The function `make-local-variable` will declare that the particular variable can be set in this buffer without affecting the value in any other buffer. Thus if we have executed the form

```
(setq foobar 5)
```

and we later execute

```
(make-local-variable 'foobar)
```

if we stay in the same buffer that we did this and type

```
(setq foobar 3)
```

we will not affect `foobar`'s value in any other buffer. `Make-local-variable` only works for the buffer in which it was executed; thus if we switched buffers and modified `foobar`'s value, it would be modified for all buffers except for those in which we did a `make-local-variable`.

Some local variables are used by low level Lisp functions, such as the code which saves files. Some of these variables are defined by higher level application programs such as RMAIL.

Buffers are full Lisp objects; they can be passed around, manipulated, etc. with a variety of commands. Some of the more useful are

- `switch-to-buffer` selects the buffer or buffer name given it. This is exactly what `C-xb` does.
- `set-buffer` makes the given buffer the current buffer for output operations, but does not select it. This can only be used inside a function to redirect output, because upon return from the command that called the function, emacs will automatically reselect the selected buffer, thus undoing this function's effects.
- `pop-to-buffer` tries to select the given buffer in another window, which is accomplished by either splitting the screen if only one window exists or placing the buffer in another window already existing on the screen. Note that window does **not** mean an X window; it means a region of the full emacs screen. `C-x2` will give you an idea of what is meant by creating a new window.
- `kill-buffer` destroys the buffer that is its argument.
- `get-buffer` returns the buffer with the name of the argument (if any). This can then be used as an arguement to another function; i. e.

  ```
  (switch-to-buffer (or (get-buffer "Foo") "*scratch*"))
  ```

  This is needed because there are some functions, such as `buffer-file-name`, which require a buffer rather than a buffer name. Many functions will accept either a buffer name or the actual buffer; I usually prefer to use the actual buffer.
- `get-buffer-create` is similar to `get-buffer`, except that if the buffer does not exist, it will be created. This is convenient; it is what `rmail-summary-mode` uses to create the summary buffer. If the buffer does not exist, it creates it; if it does, it returns it.

- `buffer-list` (not to be confused with `list-buffers`, which prints a list of all buffers for the user) returns a list of all buffers inside gnuemacs. For example, if you wanted an alist of all buffers with their associated filenames, the following form would work.

```
(mapcar '(lambda (buf) (cons (buffer-file-name buf) buf))
        (buffer-list))
```

  which then might return

```
((nil . #<buffer *scratch*>)
 ("/u9/rlk/talk" . #<buffer talk>)
 (nil . #<buffer  *Minibuf-1*>)
 ("/u9/rlk/Rmail/RMAIL" . #<buffer RMAIL>)
 ("/u9/rlk/Rmail/kin" . #<buffer kin>)
 ("/u9/rlk/Rmail/gnu" . #<buffer gnu>)
 ("/usr/athena/lib/gnuemacs/lisp/rmail.el" . #<buffer rmail.el>)
 (nil . #<buffer *Help*>)
 ("/u9/rlk/Rmail/nets" . #<buffer nets>)
 ("/u9/rlk/.allhosts" . #<buffer .allhosts>)
 ("/u9/rlk/.distlog" . #<buffer .distlog>)
 ("/u9/rlk/.distfiles" . #<buffer .distfiles>)
 ("/u9/rlk/.xtrc" . #<buffer .xtrc>)
 ("/u9/rlk/Rmail/trash" . #<buffer trash>)
 ("/u9/rlk/arrows" . #<buffer arrows>)
 (nil . #<buffer  *Minibuf-0*>)
 (nil . #<buffer  *Completions*>)
 (nil . #<buffer *mail*>)
 (nil . #<buffer *Buffer List*>))
```

  This list could then be used to determine, using `assoc`, whether a given file was in emacs, and if so what buffer was associated with it:

```
(assoc "/u9/rlk/talk"
       (mapcar '(lambda (buf)
                    (cons (buffer-file-name buf) buf))
               (buffer-list)))

==>  ("/u9/rlk/talk" . #<buffer talk>)
```

3.2. Point and Mark

Point is a Lisp object representing the location of the cursor in each buffer; thus, there is one point for each buffer. It requires special functions to access and set it, however. The function to read its value is `point`, which returns an integer representing the number of the character that the cursor is over, starting at 1. The function to set it is `goto-char`, which goes to the character given as the argument.

Mark is a special instance of an object known as a *marker*, which is something that points to a constant location in the buffer, irrespective of what text may have been added or deleted. Thus, the value of mark changes, but it follows its particular location. Programmers should normally not use the mark, as it is intended to be something used by the user; markers in the general case exist for programming. The function `set-mark` sets the mark to a given location, and `mark` returns the current value of mark.

Markers are the objects that should be used for programming purposes if you need to follow text around. Unlike most lisp objects, they should be explicitly deallocated. If a marker is not deallocated, it will stick around, which will slow down editing until the next garbage collect notices that the marker is no longer accessible and destroys it. The reason that markerss slow editing down is that each insertion or deletion of text requires that the marker be updated.

Functions used to manipulate markers:

- `make-marker` is the basic primitive used to create a new marker. The new marker is returned; it does not point to anything, nor is it associated with a buffer.
- `set-marker` sets a marker to a position in a buffer. If you don't give it a buffer or give `nil` as the buffer, it assumes the current buffer. The way to deallocate a marker is

  ```
  (set-marker marker nil)
  ```

  This doesn't destroy it per se, but prevents it from pointing to anything and thereby needing to be updated. `Move-marker` is a synonym for set-marker.
- `point-marker` creates a marker positioned at current point. This is equivalent to

  ```
  (set-marker (make-marker) (point) (current-buffer))
  ```
- `mark-marker` creates a marker pointing to the current mark in the current buffer.
- `marker-position` returns the numerical value of the place that the given marker points to.

One other thing to be aware of: normally, if you insert text when point is right over a marker, the text will be inserted after the marker. By using the function `insert-before-markers`, you can insert the text before the marker. This is useful if you are trying to insert text, but want the marker to move forward with the text rather than stay behind. You need not be over a marker in order to use this function.

3.3. Keymaps

A *keymap* is a mapping from keystrokes to commands. There are two forms of keymaps, *dense keymaps* and *sparse keymaps*. A dense keymap is a vector of 128 elements, each of which should be either a command or another keymap

A sparse keymap is a list whose car is the symbol `keymap` and whose cdr is a list of conses of character constants (integers between 0 and 127, inclusive) and commands, nil, or keymaps.

The dense representation is useful when binding a large number of keys, such as the global keymap and the various keymaps hanging off it (such as the keymap for the `esc` key), since it is more compact in this case and faster to look up in; the sparse representation is useful for keymaps that only define a few keys, such as those associated with most major modes.

The power of keymaps comes in part from the combination of their recursive definition and the fact that frequently you can pretend that they map entire key sequences. This is because most of the operations on keymaps can deal with key sequences by creating new keymaps as needed or referencing through the tree of keymaps.

Since keymaps can presently take two different forms, and since there is no guarantee that in the future there will not be more types of keymaps, you should always use the provided functions when manipulating keymaps.

Emacs maintains a notion of two separate keymaps: the *global keymap* and the *local keymap*. The main loop of emacs reads key sequences, and looks them up first in the local keymap. If it finds a command, it executes the command. If it finds another keymap, it waits for another character. If it doesn't find anything, or finds a null entry, it looks up the key sequence in the global keymap in the same manner. If it fails here, then emacs flushes the buffered keystrokes and beeps at the user. Both of these maps are local variables in each buffer; however, keymaps are normally shared. Thus, a buffer in C mode will have a different local keymap than a buffer in text mode, but two buffers in C mode will share the same local keymap, unless you explicitly copy the keymap. So if while editing in one C mode buffer you rebind a key, then the key will be rebound in all C mode buffers.

There are several especially common commands that are placed in keymaps:

- `undefined` If the keystroke or keysequence corresponding to this entry is typed, the terminal will beep at you.
- `self-insert-command` If a keystroke is bound to this command, the last keystroke typed (i. e. the one invoking `self-insert-character`) will be inserted into the buffer. This is what the normal printing ASCII characters are bound to in the global keymap.
- `nil` If an entry in a keymap is `nil`, the command used is taken from the global keymap. Thus, you can define only some of the commands but still use the rest of the normal editing commands that the user has available. This is not the same thing as binding something to `undefined`! If a key sequence is bound to `nil`, then emacs will attempt to look it up in the global keymap. If it is bound to `undefined`, then emacs will not attempt to look the sequence up in the global map, but will rather immediately signal an error.

There is another form of keymap known as a sparse keymap. A sparse keymap is used when it is desired to bind only a few keys specially. It consists of a list whose car is the symbol keymap and whose cdr is an list of (character . function) bindings. This is clearly more space-efficient when only a few keys must be rebound, but is slow if more than about 20 keys or so are being bound.

There are at any time two keymaps in use: the global and the local keymap. The algorithm for looking up a keystroke is to look in the local map first; if the keystroke is not defined there, the global map is checked.

There are several functions that should be used to manipulate keymaps:

- `make-keymap` creates a new dense keymap. It is currently equivalent to
  ```
  (make-vector 128 nil)
  ```
- `make-sparse-keymap` creates an uninitialized sparse keymap. It is currently equivalent to
  ```
  (list 'keymap)
  ```
- `copy-keymap` returns a copy of a keymap. This is commonly used when defining a major mode that should be like another major mode with a few keys changed. For example, the `scribe-mode-map` is created by copying `text-mode-map` and rebinding a few keys.

- `define-key` defines a keystroke, or keysequence, in a keymap. Its arguments consist of a keymap, a keysequence (string or character), and a definition (explained below). Note that while a sequence of keys can be defined, only the first is recorded in the keymap. For any others, the keystrokes are recursively bound to newly created keymaps as necessary, and the last keystroke is bound to the command argument. Thus, the fact that keymaps only define individual keys is hidden.
- `lookup-key` This provides a way to find the binding of a key sequence. It attempts to look up the binding of a key sequence in a given keymap.
- `suppress-keymap` -- this attempts to turn off all commands that modify the buffer, and allows numbers to act as arguments to commands without prefixing them with C-u. It is given a keymap as an argument, and it modifies the keymap to remove commands that it knows modify the buffer, which basically means anything bound to `self-insert-command`. This should be called before the desired mode-specific keys are bound. This is normally how major modes operating on read-only buffers (such as rmail mode) turn off self-inserting keystrokes and bind digits to prefix arguments. It is normally only used for modes that do not involve normal editing functions.

  Note that it does not rebind self-inserting characters to nil! It binds them to `undefined` in the local keymap, so that emacs will not attempt to look them up globally.

3.3.1. Key Definitions

Much of the power of keymaps stems from the flexibility of keystroke definition. The simplest form of a definition is an interactive function (command) or a symbol whose function definition is an interactive function. When this is encountered upon scanning the keyboard input, the function is simply called. Another simple case is that of a string, which is interpreted as a keyboard macro (the characters of the macro are read and interpreted as keyboard input).

There are two other legal values for a keystroke definition. The first is another keymap. In this case, the next keystroke available, whether from the keyboard or from a keyboard macro, is looked up in this child keymap. The final case is a cons of a keymap and a character, a form of indirect addressing. Here the cdr of the cons (the character) is looked up in the car (the keymap) and used as a substitute definition.

In all cases, if a symbol is provided, its function definition is used rather than its value.

The Emacs manual contains further information on the subject of keymaps.

3.4. Processes

There are many things that you cannot do directly in emacs that you can do by forking off a (Unix) process. The emacs-lisp side of this is also known as a *process*. Emacs lisp provides a fairly sophisticated means of control over subprocesses. One frequently used example is `M-x compile`; a more sophisticated example is the emacs server, which has a subprocess listening on a Unix domain socket.[2]

---

[2] You can think of a Unix domain socket as one end of a pipe, except that the pipe has a name in the Unix filesystem; it's a way for two processes to pass data between each other.

Many processes are associated with a buffer; their output goes into that buffer. Other processes have a *filter* associated with them; instead of being routed to a buffer, output is passed to the filter, which is a Lisp form. When output is received, it is passed to the filter. The command `M-x display-time` uses a filter that puts the time string in the mode line.

Processes are Lisp objects just like buffers and keymaps; they have more complex functions to manipulate them because they are more complex objects. Nonetheless, since they are very useful, it is worthwhile to learn how to manipulate them.

There are three commands just to create processes:

- `call-process` starts a process, and waits for it to exit. You can pass it a filename for input (the standard input for the Unix process), a buffer for output, a flag to tell it whether to display output synchronously, and arguments for the actual program.
- `call-process-region` is similar to `call-process` except that it is given a region of text in the current buffer to pass to standard input instead of an optional filename. Example (from sendmail.el):

```
(apply 'call-process-region
       (append (list (point-min) (point-max)
                     (if (boundp 'sendmail-program)
                         sendmail-program
                       "/usr/lib/sendmail")
                     nil errbuf nil
                     "-oi" "-t")
               ;; Don't say "from root" if running under su.
               (and (equal (user-real-login-name) "root")
                    (list "-f" (user-login-name)))
               ;; These mean "report errors by mail"
               ;; and "deliver in background".
               (if (null mail-interactive)
                   '("-oem" "-odb"))))
```

  The fourth argument (nil in this case) tells `call-process-region` to delete the region of text if non-nil; the sixth argument (nil) tells the function to display output as it is inserted if non-nil. Note the use of `apply` to build up the argument list.
- `start-process` simply starts a process, without waiting for it to exit. This is useful for writing “daemons” such as the emacs server, or for things such as `M-x compile` or `M-x shell`. It takes slightly different arguments; the process name, an associated buffer, the name of the executable file, and the program arguments. Example:

```
(setq compilation-process
      (start-process "compilation" "*compilation*"
                     shell-file-name
                     "-c" (concat "exec " command)))
```

  Emacs searches the directories specified by the `exec-path` variable, so you don't have to specify the complete filename.

A variety of information is associated with each process: the status (running, stopped, received a signal), the buffer, the *process sentinel*, the *filter*, and other information. The sentinel and the filter are two objects that can process information about the process: the sentinel is called every time the process changes state (e. g. stops, exits, etc.), and is passed the process and a string describing the change in state; it can then perform an operation. For example, the sentinel on the server process is called whenever the server stops; it then reads a line from the server's buffer to tell it which files to visit. The function `set-process-sentinel` sets the sentinel for a process; `process-sentinel` returns it.

The filter is called each time a batch of output from the process is available, in lieu of placing the output in a buffer. The filter is passed the process itself and the output string. `set-process-filter` sets the filter (if the filter is set to `nil` then the filter is removed and output is directed into the buffer, if any), and `process-filter` returns it.

A variety of functions are available for process control.

- `interrupt-process` sends a process an interrupt signal (SIGINT). Actually, it sends an interrupt to the process group of the process. You can tell this to send an interrupt to the process group of the process's terminal by giving a second non-`nil` argument to the function; this is used by shell mode to interrupt the current subjob of the shell rather than the shell itself.
- `kill-process` sends the process a kill signal (SIGKILL), which kills the process with no saving throw allowed. Otherwise, this is similar to `interrupt-process`.
- `stop-process` sends the process a stop signal (SIGTSTP), which normally stops the process.
- `continue-process` sends the process a continue signal (SIGCONT), which makes the process continue execution.
- `quit-process` sends the process a quit signal (SIGQUIT), which with any luck elicits a core dump from the process.
- `send-region` sends a region of text to a process.
- `delete-process` kills a process and forgets about it. This is when you REALLY want to terminate it with extreme prejudice.
- `process-send-eof` ‘‘sends’’ an end of file to a process.
- `accept-process-output` lets emacs immediately receive any output from subprocesses. Normally emacs waits for the user to be idle before accepting process output; this forces emacs to look for output when it is called.

There are also ways to get information about a process:

- `get-process` returns the process with the given name.
- `get-buffer-process` returns a process associated with the given buffer. It is possible for a buffer to have multiple associated processes.
- `process-buffer` returns the buffer associated with a process.
- `process-command` returns the command that was *exec*ed to start the process.
- `process-filter` returns the filter, if any, associated with the process.
- `process-id` returns the process id of the process.
- `process-name` returns the process with the given name.
- `process-mark` returns the marker for the end of the last output from the process. Useful for shell buffers, for example, to find the shell prompt.
- `process-sentinel` returns the sentinel, if the process has one.
- `process-status` returns a symbol representing the status of the process.

In most of these cases, it is possible to set the appropriate item (e. g. process-buffer, filter, sentinel) by means of `set-process-`<item>.

The function `process-list` can be used to get a list of all processes, analogous to `buffer-list`.

A feature recently added to Emacs is the ability to create TCP (internet stream) connections to remote hosts. These are created with `open-network-stream`, which takes four arguments: a name, associated buffer, remote host, and service port (either name or number). Once created, it looks exactly like a process created with start-process and can be manipulated as such. To the best of my knowledge, no one has yet written an application which uses this facility. This may not be anyone's loss. There is no corresponding way to create a ‘‘process’’ to listen on a port, which might be quite useful (networked emacs server?) or to create a Unix domain socket, which would make the emacs server more efficient.

The obvious use of this facility is an NNTP client to make rnews work at those sites which use this protocol to read news locally.

3.5. Syntax Tables

Syntax tables are how one specifies to emacs how certain characters should be interpreted. This is how, for example, emacs knows how to match parentheses, how to find comments, whitespace, etc. It is physically a vector of 256 integers, which encode various information about what class of character it is (i. e. whitespace, parenthesis, word-constituent, etc.), what kind of parenthesis it matches (if any), and various other information. A more detailed description can be found in the manual for those who are interested, or in the `info` system.

The most important functions for operations on syntax tables are:

- `make-syntax-table` creates a new syntax table.
- `standard-syntax-table` creates a new syntax table initialized with the default syntax information.
- `modify-syntax-entry` modifies the syntax of a character in the current syntax table. It takes rather magical arguments; if you don't understand the documentation you should probably not be using it.
- `set-syntax-table` tells emacs to use the specified syntax table in the current buffer.

4. Naming Conventions

There are certain conventions that should be followed when naming files or functions in GNU Emacs, due to the single global namespace. Some of these are enforced by the structure of the GNU Emacs directory tree, and some of these should be followed by the programmer.

4.1. Standard Filenames

Each site has an emacs tree which usually contains the lisp, info, and etc directories; the source tree, which contains the entire C source, is usually located elsewhere. All Lisp code resides in the `lisp` subdirectory; the files for the `info` system reside in the `info` subdirectory, and various auxiliary files and programs are in the `etc` directory.

Here are the important subdirectories of the Emacs tree.

4.1.1. Etc Directory

The `etc` directories contain various utilities and documents. The most useful and interesting of these are:

- `loadst` is a program that displays the time and load average in the mode line. This is invoked by

  `M-x display-time`

  Not normally invoked directly by users.

- `etags` is a very useful program to make *tags* files for gnuemacs. A tags file is a list of functions by source file along with their approximate locations. It is used by emacs to quickly find the definition of a function The files produced are not compatible with those produced by the Unix distribution program known as `ctags`. This program is very useful when maintaining a large system of C or lisp code, such as gnuemacs; it allows you to specify a function name and quickly find it. The syntax is simple:

```
etags [files...]
```

  One useful thing to do is

```
etags lisp/*.el lisp/term/*.el
```

  which makes a file called `TAGS` in the current directory for all .el files (Lisp source) in the gnuemacs distribution. This tags file can be used in several ways; it can be used to find the definitions of all functions whose name matches a given name, or it can be used to search an entire set of files for all occurrences of a given string. Examining the code in the existing Lisp libraries is one good way of discovering new ideas.

  In general, etags defines a set of files. By judicious use of the tags facilities, it's possible to define operations which work on an entire set of files rather than just a single file, such as `tags-query-replace`. This can be used even on sets of files with no source code.

- `NEWS` is a file of changes made in gnuemacs. It can be accessed by means of C-h n. Useful if you're wondering if some feature exists or why something suddenly isn't working. This also explains the latest changes to the code, and any new procedures that programmers should follow. This file is authoritative, and supersedes anything contained in this document.

- `DOC` contains the documentation for the preloaded functions in gnuemacs. Not interesting to look at, but it doesn't hurt to know that it exists.

- `GNU` is RMS's GNU manifesto.

- `refcard` is the nroff/troff source for an emacs reference card.

4.1.2. Lisp directory

Lisp code is located here, in both the source and *byte-compiled* form. Byte compiling is an operation that compacts the Lisp code and makes it load and execute faster; it is not compiling into raw executable code.

`M-x byte-compile-file`

compiles a given file;

`M-x byte-recompile-directory`

recompiles the .el (source) files into .elc (compiled) files as necessary. It only tries to compile files that have older .elc files; a .el file without a .elc file will not be byte compiled as there are certain files that should not be byte compiled.

The lisp code is a fundamental part of gnuemacs; it is not separate in any way, shape, or form.

- `~/.emacs` (the user's .emacs file) While this isn't in the lisp directory, it is the most important lisp file from the user's point of view. It is the user's way of customizing the environment. A sample .emacs file has been provided as an example; it is actually the conglomeration of the .emacs files of a number of Project Athena staff and SIPB members. It typically contains code to set variables, define keystrokes, define any special functions that the user may have written, etc. Making it too big will make emacs take too long to start up; it is preferable to create other libraries and load or autoload them when needed.

- `term directory` This contains files of Lisp code that are loaded to customize emacs for specific terminal types or window systems, such as xterm, vt100, vt220, etc.

- `loadup.el, loaddefs.el, version.el, grow-vers.el` are special files used only during the building of an emacs. They only rarely need to be modified by hand, and even then loaddefs should be the only one modified unless you REALLY know what you are doing. Loaddefs contains defvars for most user option variables except for those which must be defined at runtime, which are generally either in their respective files, in startup.el, or in site-init.el. Note that these files are NOT byte-compiled.

- `site-init.el` contains site-specific code that is loaded at build time, and thus becomes part of the emacs lisp world. A similar file is `site-load.el`; this should only contain forms to load desired libraries, as it is loaded at a different point in the load process.

- `default-profile.el` is a default .emacs file that is loaded after the user's .emacs file. This can be inhibited by setting `inhibit-default-init` to `t`. The difference between this and `site-init.el` is that this file is loaded at run time, rather than load time.

- `startup.el` is the file that is loaded upon startup of an emacs. This does a variety of tasks such as processing command-line arguments, loading your .emacs file, and starting the main command loop.

- Files controlling special major modes -- the files of code for all major modes other than fundamental, which has no special code of its own, are in files of the form <foo>-mode.el and .elc. Thus, the file of code for `text-mode` is in `text-mode.el`, that for lisp mode is in `lisp-mode.el`, etc. However, in order that the filenames be acceptable to losing versions of emacs that restrict pathname components to 14 characters, some of the names are truncated, such as `picture.el` which implements picture mode.

- Files implementing special forms of ''editing'' -- not all of the lisp code is meant for editing files for programming or text processing. There are two packages for sending and receiving mail, Rmail (which consists of a number of files, actually), which is the Unix program implementing Babyl file format, as used in Tops-20 and ITS Babyl programs and Zmail on the lisp machine, and mh-rmail, which is present in `mh-e.el` and implements a more pleasant user interface to the *mh* mail system. There is also an implementation of Eliza, `doctor.el`, and the Tower of Hanoi, in `hanoi.el`, and others. These two (Eliza and Hanoi) demonstrate the power of gnuemacs lisp to perform tasks that ''normal'' lisps can perform.

4.2. Function naming conventions

In general, functions should be named so that they describe what they do. You should attempt to use ''keywords'' so that `M-x apropos` will find it. Thus, if you are writing a function that will munge paragraphs, the name should have the word `paragraph` in it. If you are writing a new major mode, all functions that are a part of that mode, at least those that other programmers should be able to use, should have the name of the mode embedded in their name, preferably as the first word or words of the function or variable name. Thus, all functions that are explicitly a part of rmail are named `rmail-`*function-name*.

4.3. Keystroke binding conventions

There are several keystroke naming conventions that should be used when binding keys to functions in new major modes. These need not be followed in personal customizations, but should be observed in all code that is intended for distribution. These change from time to time, generally becoming tighter. You should check `etc/NEWS` from time to time to learn the latest changes. Here are the current conventions.

- Do not bind anything to `C-z`. This key is reserved at all times to suspend-emacs. `C-xC-z` is also reserved for this purpose, and `C-xC-c` should also not be used, as it is reserved for exiting emacs. `M-x` should of course never be rebound, and `M-esc` should not either, so the user can evaluate Lisp forms in the minibuffer.
- The commands beginning with `C-c` are reserved for special major modes. These characters are ideal for special commands that are not meaningful when editing ordinary text. However, two-keystroke commands beginning with C-c and ending with a single ASCII letter are reserved to the user for personal customization.
- Command names should parallel keystrokes for simpler commands. Thus, commands that have the effect of moving forward in some sense of the term (such as to the next message in rmail) should use commands with `f` or `n`, or `C-f` or `C-n`, etc. This makes it much easier for users to remember commands with analogous actions between modes.
- Command names should be mnemonic when possible. Thus, if you have a choice of keystrokes, you should use ones that provide a clue about what they do. Thus, commands with `f` in them generally move forward some unit of text, etc.

5. Major Customization

The way to make major customizations to your emacs environment it to sit down and write Lisp code for the editor. The rest of this talk will be concerned with this.

Many examples will be taken from rmail, since I am very familiar with that particular package and it is one of the more elaborate systems inside emacs. However, I will use other examples as well.

5.1. The .emacs File

The `.emacs` file is the basic way of customizing your emacs environment. Simple .emacs files will contain settings for variables; more complicated ones will contain definitions for significant numbers of functions. A sample .emacs file, heavily documented, is provided.

However, you may find yourself writing a package that takes a long time to load, and is reasonably coherent. In this case, you should place this material in a separate file and compile it, and consider submitting it to the Free Software Foundation.

5.2. Major Modes

If you are writing large quantities of code to perform specialized editing tasks, you probably want to write a new major mode. Several examples of the code needed to implement major modes are provided. Rmail mode is rather complicated, for example. Rmail-edit-mode, on the other hand, is a very simple major mode that is very similar to text mode except that it includes a few additional keystroke commands.

Some jobs, such as rmail edit, would seem to call for a recursive edit. Rmail edit is a mode which allows the user to edit the current rmail message, so it makes sense to ‘‘recursively’’ edit the current message. Nevertheless, it is better to write a new major mode, since a recursive edit is confusing to the user. The only times that a recursive edit should be used is when there is no other way to implement something. One example of this is the Emacs debugger.

5.3. The Debugger

The debugger works by actually modifying the function definition of the functions that it debugs. For this reason, it can only work on functions defined in the interpreted or byte-compiled Lisp environment, as opposed to hard-coded primitives. Printing the function definition of a function, using symbol-function on the name of the function, will either print out a list (the interpreted or byte-compiled code) or something of the form `#<subr foo>`, which indicates a compiled function. The list comprising the lisp-encoded function can obviously be modified; the subroutine cannot be.

The debugger itself is slightly arcane, but is not too difficult to learn. The only way to learn how to use it is to play with it. It can be invaluable when debugging an obscure bug.

5.4. Minor Modes

Minor modes are used when implementing a minor modification that should be accessible in many major modes, such as auto-fill or auto-justify. The biggest problem is finding a way to insert the necessary hook into the running system. The auto-fill-hook already exists; it works for auto-justify-mode because the two functions are very similar. However, for a minor mode that works very differently, writing the code is more difficult. The principal problem is that self-insert-command, the primitive to insert the last key typed, is a hard-coded subroutine. Here is one way of doing it:

Minor mode hacking is much more difficult and normally less useful than major mode hacking.

I. Some Examples

I.1. A .emacs File

This is a sample .emacs file. It's rather longer than any sane individual would want, but it gives a flavor of the sort of things that people would put in their .emacs file. It is somewhat out of date in some minor respects.

```
;;;     $Source: /mit/r/l/rlk/e17/RCS/world.emacs.el,v $
;;;     $Author: rlk $
;;;     $Locker:  $
;;;     $Header: world.emacs.el,v 1.3 86/01/21 14:07:38 rlk Exp $

;;;.emacs.el file -- compiled to .emacs.elc by "byte-compile-file"
;;;
;;; Some global data

;;; The load path is a variable that controls where Lisp files will be
  loaded
;;; in the same fashion that the PATH variable controls where the
;;; shell will search.
;;;

;;; Note the new mode line format in Version 18.

(if (string-match "^18" emacs-version)
    (setq system-name (system-name)
          default-mode-line-format
          '("--%1*%1*-"
            system-name
            ":"
            emacs-version
            " "
            (-17 . mode-line-buffer-identification)
            "   "
            global-mode-string
            "   %[("
            mode-name
            minor-mode-alist
            "%n"
            mode-line-process
            ")%]----"
            (-3 . "%p")
            "-%-")
          mode-line-format default-mode-line-format
          load-path (append load-path
                            '("/mit/r/l/rlk/em"
                              "/mit/rlk/em"
                              "/@/priam/mit/r/l/rlk/em"
```

```
                                "/usr/unsupported/watchmkr/lib/gnuemacs/lisp
  (setq mode-line-format (concat "--%1*%1*-" (system-name)
                                 ": %17b   %M   %[(%m)%]----%3p-%-")
        default-mode-line-format mode-line-format
        load-path '(nil
                    "/mit/r/l/rlk/em"
                    "/mit/rlk/em"
                    "/@/priam/mit/r/l/rlk/em"
                    "/usr/unsupported/watchmkr/lib/gnuemacs/lisp"
                    "/usr/athena/lib/gnuemacs/lisp"
                    "/usr/mit/lib/gnuemacs/lisp"
                    "/usr/local/emacs/lisp")))

(setq mh-progs "/usr/new/mh.6/bin")     ; Directory containing MH commands

(setq mh-lib "/usr/new/mh.6/lib")       ; Directory of MH library

;;; This is a special function to take any number of arguments and do
;;; nothing whatsoever.  It is useful on occasion.
(defun nop (&rest ignore) (interactive))

;;;This variable controls the appearance of the mode line.  It is a
;;;rather complicated example of formatting specifications.  Do C-h v
;;;on mode-line-format to find out more information.

;;; This presumably is for a user-defined function to flush help
;;; windows.
(setq remove-help-window t)

(setq display-time-day-and-date t)

;;;
;;; Functions
;;;courtesy Jim Fulton

(defun my-kill-help-window ()
  "Kill off the stupid help window."
  (interactive)
  (delete-windows-on "*Help*"))

;;; This is an example of a minor mode.  Note the interactive
;;; specification.  The auto-fill-hook is called whenever a
;;; whitespace character is typed when a line spills over the fill
;;; column.  This is similar to normal auto-fill-mode except that it
;;; additionally justifies paragraphs.

(defun auto-justify-mode (arg)
  "Toggle auto-justify mode or turn auto-fill mode on iff arg is positive.
In auto-fill mode, inserting a space at a column beyond  fill-column
automatically breaks the line at a previous space. Also automatically
justifies paragraphs."
  (interactive "P")
  (setq auto-fill-hook
        (and (if (null arg)
                 (not auto-fill-hook)
               (> (prefix-numeric-value arg) 0))
             '(lambda ()
```

```
                (fill-paragraph t)
                (insert " "))))
  (set-minor-mode 'auto-justify-mode "Justify"
                  (not (null auto-fill-hook))))

;;; Note the documentation construct for this function.  The \[...\]
;;; construct tells the documentation function to insert the key
;;; sequence (keystroke or M-x) to invoke this function

(defun make ()
  "Synonym for \\[compile\] for people who prefer this name."
  (interactive)
  (call-interactively 'compile))

;;; Example function produced by use of a named keyboard macro.

(fset 'cpr "/* Copyright (c) 1985 Massachusetts Institute of Technology */")

;;; Example of a normal lisp primitive that does not exist in the
;;; built system that can be defined easily in emacs-lisp.  Note the
;;; iterative definition as opposed to a more normal recursive definition.

(defun member (elt list)
  "Returns non-nil if ELT is an element of LIST.  Comparison done with
  equal.
The value is actually the tail of LIST whose car is ELT."
  (while (and y (not (equal x (car y))))
    (setq y (cdr y)))
  y)

(defun count-words-in-buffer ()
  "Counts the words in the current buffer, displaying the current count,
along with a timestamp and the buffer name, in the *Word Count* buffer."
  (interactive)
  (count-words-in-region (point-min) (point-max)))

;;; This is the standard way of passing the region to a function.  A
;;; higher level function, such as count-words-in-buffer, can call
;;; this without smashing the mark.  It exhibits process control,
;;; using the call-process-region function, buffer manipulation, and
;;; output to a buffer.  This is a classic example of a
;;; properly-written emacs-lisp function -- it performs a single job
;;; with a simple interface that can be used from either the keyboard
;;; or another lisp function.  Note that it prepends the word count
;;; to the word count buffer.  This provides a way of watching the
;;; growth of a file.

(defun count-words-in-region (min max)
  "Counts the words in the current region, displaying the current count,
along with a timestamp and the buffer name, in the *Word Count* buffer.
If called from a program, expects MIN and MAX args as start and end
buffer addresses."
  (interactive "r")
  (let ((real-buffer (current-buffer))
        (buffer (get-buffer-create "*Word Count*")))
    (set-buffer real-buffer)
    (call-process-region min max "/usr/ucb/wc" nil buffer nil)
```

```
    (set-buffer real-buffer)
    (princ (buffer-name) buffer)
    (princ " (region) as of " buffer)
    (princ (current-time-string) buffer)
    (terpri buffer)
    (pop-to-buffer buffer)
    (goto-char (point-min))
    (pop-to-buffer real-buffer)))

;;; A simple interface to an important variable.  This function
;;; provides a way for the user to do something simple (set tab stops
;;; every n columns) on a variable that is fairly complicated to set.
;;; Useful and simple.  This can also be called from a program.

(defun set-tabs (n)
  "Sets tabs every N columns."
  (interactive "p")
  (let ((count n) (tabs nil))
    (while (<= count 120)
      (setq tabs (cons count tabs))
      (setq count (+ count n)))
    (setq tab-stop-list (reverse tabs))))

;;; Note that this simple function signals an error if the buffer is
;;; read-only.  A proper use of interactive *.

(defun newline-and-indent-relative ()
  "Insert a newline, then indent relative to the previous line."
  (interactive "*")
  (let ((hpos (current-indentation)))
    (delete-region (point)
                   (progn (skip-chars-backward " \t")
                          (point)))
    (insert ?\n)
    (indent-relative)))

(defun fork-shell (bufname)
  "Run an inferior shell, with I/O through buffer *BUFNAME*
If buffer exists but shell process is not running, make new shell.
Program used is taken from ESHELL environment variable,
 or else from SHELL if there is no ESHELL.
If a file ~/.emacs_SHELLNAME exists, it is given as initial input
 (Note that this may lose due to a timing error if the shell
  discards input when it starts up.)
The buffer is put in shell-mode, giving commands for sending input
and controlling the subjobs of the shell.  See shell-mode.
See also variable shell-prompt-pattern.

Note that many people's .cshrc files unconditionally clear the prompt.
If yours does, you will probably want to change it."
  (interactive "sShell buffer: ")
  (let* ((prog (or (getenv "ESHELL") (getenv "SHELL")))
         (name (file-name-nondirectory prog)))
    (switch-to-buffer
     (make-shell bufname prog
                 (if (file-exists-p (concat "~/.emacs_" name))
                     (concat "~/.emacs_" name))))))
```

```
(defun tabify-whole-buffer ()
  "Runs tabify on the current buffer."
  (interactive)
  (tabify (point-min) (point-max)))

;;; This provides a way for minibuffers that use completion to exit
;;; without the user having to type a newline, or otherwise finish
;;; the completion.  This is an example of minibuffer manipulation.

(defun minibuffer-complete-exit-backup ()
   "Minibuffer completion, exiting on unique completion with backup."
   (interactive)
   (let (comp (complete t))
     (while
         (null
          (setq comp
                (all-completions
                 (buffer-substring (point-min) (point))
                 minibuffer-completion-table
                 minibuffer-completion-predicate)))
       (setq complete nil)
       (delete-backward-char 1 nil))
     (and complete (if (null (cdr comp))
                        (minibuffer-complete-and-exit)
                      (minibuffer-complete)))))

;;; A simple, yet fairly clever use of define-key.  Since define-key
;;; returns the function definition, it is possible to use the return
;;; value in another define-key.  It isn't the most transparent
;;; coding style, though, and probably should be avoided.

(define-key minibuffer-local-must-match-map " "
     (define-key minibuffer-local-completion-map " "
          'minibuffer-complete-exit-backup))

;;; More process manipulation.  This shows some more complicated
;;; manipulations of processes running under gnuemacs asynchronously.

(defun syslog ()
  "Tail /usr/spool/mqueue/syslog in buffer *Syslog*"
  (interactive)
  (require 'shell)
  (let ((buffer (get-buffer-create "*Syslog*")) proc status)
    (setq proc (get-buffer-process buffer))
    (if proc
        (setq status (process-status proc)))
    (save-excursion
      (set-buffer buffer)
      (if (memq status '(run stop))
          nil
        (if proc (delete-process proc))
        (setq proc (start-process "Syslog" buffer
                                  "/usr/ucb/tail" "-f"
                                  "/usr/spool/mqueue/syslog")))
      (shell-mode))
    (switch-to-buffer buffer)))
```

```
;;;
;;; Don’t coddle me.

(setq inhibit-startup-message t)        ; i’ve already read it

;;; Many people prefer to start up in lisp interaction mode, to work
;;; in a lisp listener.

(setq startup-major-mode ’lisp-interaction-mode)

(setq default-major-mode ’text-mode)    ; No hooks for fundamental mode

;;; Enabling some commands.  Note that the disabled property is
;;; implemented by use of the property list of the symbol.

(put ’eval-expression ’disabled nil)    ; <Esc> <Esc> or Meta-<Esc>

(put ’narrow-to-region ’disabled nil)

(put ’rmail ’disabled nil)

;;; More personal customizing of the environment.

(setq enable-recursive-minibuffers t)

(display-time)

;;;
;;; X configuration

;;; This is an example of conditional customization.  Note that one
;;; thing is done if xterm is non-nil, and something else is done if
;;; it is.  There is nothing wrong with having non-defuns at top level.

(if xterm
    (progn (x-set-font "6x10")
           (global-set-key "\^z" ’nop)
           (global-set-key "\^x\^z" ’nop)
           (x-set-bell t))              ; Set X terminals for flash, else
  (setq visible-bell nil)) ; no visible bell

;;;
;;; Mail handling

(setq mail-self-blind t)                ; Bcc to me

;;; A fairly complicated example of a hook being used to customize a
;;; major mode.  This one inserts a Cc: if necessary, along with some
;;; other headers, and places the cursor in the To: field.  It will
;;; be called whenever a piece of mail is being initialized.

(setq mail-setup-hook                   ; When sending mail
      ’(lambda ()
         (if to                         ; If reply, point is below header
             (forward-line -1)          ; Move back before separator.
           (forward-line 2))            ; New Mail, skip To: and Subject:
         (if (not cc) (insert "Cc: \n")); Add a cc if not there.
```

```
          (insert
           "Full-Name: David G. Grubbs\n"        ; Sendmail uses this.
           "Reply-To: dgg@ATHENA.MIT.EDU\n" ; Return address
           "US-Snail: Room E40-442B, 1 Amherst St, Cambridge, MA 02139\n")
          (if to
              (goto-char (point-max))    ; If a reply, move below headers
            (goto-char (point-min))
            (re-search-forward "^To: ")))) ; Else go to the To: line

;;; These are several miscellaneous variables that this user wished to set.

(setq rmail-delete-after-output t)      ;get rid of messages when refiled

(setq rmail-dont-reply-to-names "dgg\\|root\\|info")

(setq rmail-file-name (expand-file-name "~/RM/RMAIL"))

(setq rmail-last-file (expand-file-name "~/RM/mbox")
      rmail-last-rmail-file (expand-file-name "~/RM/history"))

;;; Note that this variable’s value is a regular expression.

(setq rmail-ignored-headers
      (concat "^via:\\|^mail-from:\\|^origin:\\|^status:\\|^received:"
              "\\|^message-id:\\|^summary-line:\\|^return-path:\\|"
              "^.*-of-the-year:\\|^.*ism:\\|^precedence:" )

;;;
;;; File Handling

;;; These all customize the internal operations of the editor.  If you
;;; wish to find out what these do, you should read the documentation
;;; on the variables.

(setq backup-before-writing t)

(setq backup-by-copying-when-linked t)

(setq auto-mode-alist
      (cons ’(("\\.txt$" . text-mode)
              ("\\.mss$" . text-mode)
              ("\\.qc$"  . c-mode)
              ("\\.clu$" . fundamental-mode)
              ("\\.el$"  . emacs-lisp-mode)
              ("\\.q$"   . c-mode)
              ("\\.qh$"  . c-mode))
            auto-mode-alist))

(setq completion-ignored-extensions
   ’(".o" ".elc" "~" ".ckp" ".bak" ".imp" ".lpt" ".bin" ".otl" ".err"
  ".lib"))

(setq delete-auto-save-files t)         ;I don’t want cruft

(setq list-directory-brief-switches "-aCF"); for completeness

(setq list-directory-verbose-switches "-lagF")
```

```
(setq version-control t)                ;to give version number backups

;; Define some hooks (mode customizing functions). Note that defun will not
;; work (use lambda) because defun shoves code into function slot of atom,
;; while setq shoves into value slot. All modes check the value slot for
  hooks.
;; Alternatively, you can use (setq <mode name> (defun <mode name> ...))
;; but this is wasteful.

;;; These are typical hooks that you would want to set up to handle
;;; specific modes.  These merely set some variables.

(setq c-mode-hook '(lambda ()
                     (setq c-indent-level 4
                           c-argdecl-indent 0
                           comment-column 40
                           c-label-offset -4
                           c-continued-statement-indent 0
                           c-brace-offset 0)))

(setq lisp-mode-hook '(lambda ()
                        (setq comment-start "; ")
                        ;; This means that the lisp-indent-hook
                        ;; property of lambda is now defun.  It so
                        ;; happens that lisp-mode interprets it to
                        ;; mean that lambda should be interpreted as a
                        ;; defun for indentation purposes.  Property
                        ;; lists are a good place to store this kind
                        ;; of information.
                        (put 'lambda 'lisp-indent-hook 'defun)
                        (setq comment-column 40)
                        (local-set-key "\n" 'newline-and-indent)))

(setq text-mode-hook '(lambda ()
                        (setq fill-column 75)
                        (auto-fill-mode 1)))


;;;
;;; Key bindings

;;; One way of forming multi-stroke keybindings.  Note the use of
;;; esc-i globally to mean insert-file

(define-key esc-map "#" 'count-words-in-region)
(define-key esc-map "G" 'goto-line)
(define-key esc-map "R" 'query-replace)
(define-key esc-map "s" 'spell-word)
(define-key esc-map "S" 'spell-buffer)
(define-key esc-map "T" 'tags-search)
(define-key esc-map "W" 'what-line)
(define-key esc-map "o" 'open-rectangle)
(define-key esc-map "i" 'insert-file)
(define-key help-map "A" 'apropos-command)
(define-key help-map "a" 'apropos)
(define-key help-map "\^k" 'kill-help-window)
(define-key ctl-x-map "!" 'fork-shell)
```

```
(define-key ctl-x-map "#" 'count-words-in-buffer)
(define-key ctl-x-map "T" 'tabify-whole-buffer)
(define-key ctl-x-map "f" 'auto-fill-mode)
(define-key ctl-x-map "t" 'transpose-lines)
(define-key ctl-x-map "r" 'rmail)
(define-key ctl-x-map "\^b" 'buffer-menu)
(define-key ctl-x-map "\^n" 'next-error)
(define-key global-map "\^?" 'apropos)

;;; Another way.  This will be used only in text mode.  What will
;;; happen is that a new sparse keymap will be defined that will take
;;; the esc key and map the next character.

(define-key text-mode-map "\ei" 'indent-relative)
(define-key text-mode-map "\^j" 'newline-and-indent-relative)
```

I.2. X Window System Code

This is the old version of the file term/xterm.el. The newer version has all the interesting stuff cut out of it and moved into the C code.

```
;;;Example of proper use of defvar.  If the user has already set the
;;;variable, then his value won't get mashed.  Else we use the default.

(defvar x-switches nil
  "Alist of command switches and values for X window system interface.
You can set this in your init file, if you want some defaults
for these switches.  Example:
  (setq x-switches '((\"-r\" . t) (\"-font\" . \"foo\") (\"-b\" . \"8\")))")

;; The special code for X (xterm.c) depends on using interupt-driven input.
(set-input-mode t nil)

;; Not defvar!  This is not DEFINING this variable, just specifying
;; a value for it.
(setq term-setup-hook 'x-pop-up-window)

(setq command-switch-alist
      (append '(("-r" . x-handle-switch)
                ("-i" . x-handle-switch)
                ("-font" . x-handle-switch)
                ("-w" . x-handle-switch)
                ("-d" . x-handle-switch)
                ("-b" . x-handle-switch)
                ("-fg" . x-handle-switch)
                ("-bg" . x-handle-switch)
                ("-bd" . x-handle-switch)
                ("-cr" . x-handle-switch)
                ("-ms" . x-handle-switch))
              command-switch-alist))

;; This is run after the command args are parsed.
(defun x-handle-switch (switch)
  (if (x-handle-switch-1 switch (car command-line-args))
      (setq command-line-args (cdr command-line-args))))

;;; Essentially a large switch statement.  This is a typical use for
;;; cond.  Else we could have used the command-switch-alist
;;; differently, perhaps.

(defun x-handle-switch-1 (switch arg)
  (cond ((string= switch "-r")
         (x-flip-color)
         nil)
        ((string= switch "-i")
         (x-set-icon t)
         nil)
        ((string= switch "-font")
         (x-set-font arg)
         t)
        ((string= switch "-b")
         (x-set-border-width (string-to-int arg))
         t)
```

```
          ((string= switch "-d")
           (x-new-display arg)
           t)
          ((string= switch "-w")
           (x-create-x-window arg)
           t)
          ((string= switch "-fg")
           (x-set-foreground-color arg)
           t)
          ((string= switch "-bg")
           (x-set-background-color arg)
           t)
          ((string= switch "-bd")
           (x-set-border-color arg)
           t)
          ((string= switch "-cr")
           (x-set-cursor-color arg)
           t)
          ((string= switch "-ms")
           (x-set-mouse-color arg)
           t)))

;; Convert a string of the form "WWxHH+XO+YO",
;; where WW, HH, XO and YO are numerals,
;; into a list (WW HH XO YO).
;; "xHH" may be omitted; then 0 is used for HH.
;; XO and YO may be preceded by - instead of + to make them negative.
;; Either YO or both XO and YO may be omitted; zero is used.

;;; Typical use of string handling functions to parse something.  This
;;; should give ideas about writing filters and the like.  We could
;;; easily reassemble this into another buffer, if that was what we
;;; were interested in.

(defun x-parse-edge-spec (arg)
  (let ((cols-by-font 0)
        (rows-by-font 0)
        (xoffset 0)
        (yoffset 0))
    (if (string-match "^=" arg)
        (setq cols-by-font (x-extract-number))
      (error "Invalid X window size/position spec"))
    (if (string-match "^x" arg)         ;get rows-by-font
        (setq rows-by-font (x-extract-number)))
    (if (string-match "^[-+]" arg)
        (setq xoffset (x-extract-number)))
    (if (string-match "^[-+]" arg)
        (setq yoffset (x-extract-number)))
    (or (equal arg "")
        (error "Invalid X window size/position spec"))
    (list cols-by-font rows-by-font xoffset yoffset)))

;; Subroutine to extract the next numeral from the front of arg,
;; returning it and shortening arg to remove its text.
;; If arg is negative, subtract 1 before returning it.

;;; Note that instead of writing a huge routine, it is often better to
```

```
;;; split things up into smaller chunks.

(defun x-extract-number ()
  (if (string-match "^[x=]" arg)
      (setq arg (substring arg 1)))
  (or (string-match "[-+]?[0-9]+" arg)
      (error "Invalid X window size/position spec"))
  (prog1
      (+ (string-to-int arg)
         (if (string-match "^-" arg) -1 0))
    (setq arg
          (substring arg
                     (or (string-match "[^0-9]" arg 1)
                         (length arg))))))

;;; Another way of writing a selector.

(defun x-get-default-args ()
  (let (value)
    (if (not (string= (setq value (x-get-default "bodyfont")) ""))
        (x-handle-switch-1 "-font"  value))
    (if (string-match "On" (x-get-default "reversevideo"))
        (x-handle-switch-1 "-r" t))
    (if (string-match "On" (x-get-default "bitmapicon"))
        (x-handle-switch-1 "-i" t))
    (if (not (string= (setq value (x-get-default "borderwidth")) ""))
        (x-handle-switch-1 "-b" value))
    (if (not (string= (setq value (x-get-default "foreground")) ""))
        (x-handle-switch-1 "-fg" value))
    (if (not (string= (setq value (x-get-default "background")) ""))
        (x-handle-switch-1 "-bg" value))
    (if (not (string= (setq value (x-get-default "border")) ""))
        (x-handle-switch-1 "-bd" value))
    (if (not (string= (setq value (x-get-default "cursor")) ""))
        (x-handle-switch-1 "-cr" value))
    (if (not (string= (setq value (x-get-default "mouse")) ""))
        (x-handle-switch-1 "-ms" value))))

(defun x-new-display (display)
  "This function takes one argument, the display where you wish to
continue your editing session.  Your current window will be unmapped and
the current display will be closed.  The new X display will be opened and
the rubber-band outline of the new window will appear on the new X display."
  (interactive "sDisplay to switch emacs to:  ")
  (x-change-display display)
  (x-get-default-args))

;;;It's reasonable to load one file from another; the other arguments
;;;make it silent.

(load "x-mouse" t t)

;;; Yup, here we want to disable something.

(put 'suspend-emacs 'disabled t)
(global-set-key "\C-z" 'undefined)
(global-set-key "\C-x\C-z" 'undefined)
```

```
(x-get-default-args)

;; If used has .Xdefaults file, process switch settings for Emacs
;; recorded there.
;;(if user
;;    (let ((Xdefs-file (concat "~" user "/.Xdefaults")))
;;      (if (file-exists-p Xdefs-file)
;;        (x-get-default-args Xdefs-file))))

;; Process switch settings made by .emacs file.
(while x-switches
  (x-handle-switch-1 (car (car x-switches)) (cdr (car x-switches)))
  (setq x-switches (cdr x-switches)))
```

This is the file lisp/x-mouse.el. This is the Lisp side of the mouse interface for the X window system. The amount of support at the C level is minimal. This is an excellent example of the power of Emacs Lisp; a few primitives support a major package of Lisp code.

```
;;; Note that these are considered constants, and that programs will
;;; not change them.  Contrast with defvar.

(defconst x-button-right (char-to-string 0))
(defconst x-button-middle (char-to-string 1))
(defconst x-button-left (char-to-string 2))

(defconst x-button-s-right (char-to-string 16))
(defconst x-button-s-middle (char-to-string 17))
(defconst x-button-s-left (char-to-string 18))

(defconst x-button-m-right (char-to-string 32))
(defconst x-button-m-middle (char-to-string 33))
(defconst x-button-m-left (char-to-string 34))

(defconst x-button-c-right (char-to-string 64))
(defconst x-button-c-middle (char-to-string 65))
(defconst x-button-c-left (char-to-string 66))

(defconst x-button-m-s-right (char-to-string 48))
(defconst x-button-m-s-middle (char-to-string 49))
(defconst x-button-m-s-left (char-to-string 50))

(defconst x-button-c-s-right (char-to-string 80))
(defconst x-button-c-s-middle (char-to-string 81))
(defconst x-button-c-s-left (char-to-string 82))

(defconst x-button-c-m-right (char-to-string 96))
(defconst x-button-c-m-middle (char-to-string 97))
(defconst x-button-c-m-left (char-to-string 98))

(defconst x-button-c-m-s-right (char-to-string 112))
(defconst x-button-c-m-s-middle (char-to-string 113))
(defconst x-button-c-m-s-left (char-to-string 114))

;;; A somewhat questionable use of minor modes.  The latest version
;;; dispenses with the minor mode.

(defun x-mouse-mode ()
  "Process all queued mouse events."
  ;; A mouse event causes a special character sequence to be given
  ;; as keyboard input.  That runs this function, which process all
  ;; queued mouse events and returns.
  (interactive)
  (unwind-protect
      (progn
        (set-minor-mode 'x-mouse-mode "Mouse" t)
        (while (> (x-mouse-events) 0)
          (x-proc-mouse-event)))
    (set-minor-mode 'x-mouse-mode "Mouse" nil)))

(define-key global-map "\C-c\C-m" 'x-mouse-mode)
```

```
;;; Some rather hairy window hacking that does very little.

(defun x-mouse-select (arg)
  "Select Emacs window the mouse is on."
  (let ((start-w (selected-window))
        (done nil)
        (w (selected-window))
        (rel-coordinate nil))
    (while (and (not done)
                (null (setq rel-coordinate
                            (coordinates-in-window-p arg w))))
      (setq w (next-window w))
      (if (eq w start-w)
          (setq done t)))
    (select-window w)
    rel-coordinate))

;;; Some things are nuked here to save space.

;;; Somewhat fancier keymap hacking.  The primitive x-proc-mouse-event
performs the lookup.  There’s no particular reason that
x-proc-mouse-event needs to be a primitive, either.

(define-key mouse-map x-button-right ’x-mouse-select)
(define-key mouse-map x-button-left ’x-mouse-set-mark)
(define-key mouse-map x-button-c-s-right ’x-mouse-keep-one-window)
(define-key mouse-map x-button-c-right ’x-mouse-select-and-split)
(define-key mouse-map x-button-middle ’x-mouse-set-point)
(define-key mouse-map x-button-s-middle ’x-cut-text)
(define-key mouse-map x-button-s-right ’x-paste-text)
(define-key mouse-map x-button-c-middle ’x-cut-and-wipe-text)
```

Here's one more example from the X interface. One minor change to the underlying X code was all that was needed to implement a system of menus.

```
;; Copyright (C) 1986 Free Software Foundation, Inc.

;; This file is part of GNU Emacs.

;; GNU Emacs is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY.  No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing.  Refer to the GNU Emacs General Public
;; License for full details.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; GNU Emacs General Public License.   A copy of this license is
;; supposed to have been given to you along with GNU Emacs so you
;; can know your rights and responsibilities.  It should be in a
;; file named COPYING.  Among other things, the copyright notice
;; and this notice must be preserved on all copies.


(defmacro caar (conscell)
  (list 'car (list 'car conscell)))

(defmacro cdar (conscell)
  (list 'cdr (list 'car conscell)))

(defun x-menu-mode ()
  "Major mode for creating permanent menus for use with X.
These menus are iimplemented entirely in Lisp; popup menus, implemented
with x-popup-menu, are implemented using XMenu primitives."
  (make-local-variable 'x-menu-items-per-line)
  (make-local-variable 'x-menu-item-width)
  (make-local-variable 'x-menu-items-alist)
  (make-local-variable 'x-process-mouse-hook)
  (make-local-variable 'x-menu-assoc-buffer)
  (setq buffer-read-only t)
  (setq truncate-lines t)
  (setq x-process-mouse-hook 'x-menu-pick-entry)
  (setq mode-line-buffer-identification '("MENU: %32b")))

(defvar x-menu-max-width 0)
(defvar x-menu-items-per-line 0)
(defvar x-menu-item-width 0)
(defvar x-menu-items-alist nil)
(defvar x-menu-assoc-buffer nil)

(defvar x-menu-item-spacing 1
  "*Minimum horizontal spacing between objects in a permanent X menu.")

(defun x-menu-create-menu (name)
  "Create a permanent X menu.  Returns an item which should be used as a
menu object whenever referring to the menu."
  (let ((old (current-buffer))
        (buf (get-buffer-create name)))
```

```
    (set-buffer buf)
    (x-menu-mode)
    (setq x-menu-assoc-buffer old)
    (set-buffer old)
    buf))

(defun x-menu-change-associated-buffer (menu buffer)
  "Change associated buffer of MENU to BUFFER.  BUFFER should be a buffer
object."
  (let ((old (current-buffer)))
    (set-buffer menu)
    (setq x-menu-assoc-buffer buffer)
    (set-buffer old)))

(defun x-menu-add-item (menu item binding)
  "Adds to MENU an item with name ITEM, associated with BINDING.
Following a sequence of calls to x-menu-add-item, a call to x-menu-compute
should be performed before the menu will be made available to the user.

BINDING should be a function of one argument, which is the numerical
button/key code as defined in x-menu.el."
  (let ((old (current-buffer))
        elt)
    (set-buffer menu)
    (if (setq elt (assoc item x-menu-items-alist))
        (rplacd elt binding)
      (setq x-menu-items-alist (append x-menu-items-alist
                                       (list (cons item binding)))))
    (set-buffer old)
    item))

(defun x-menu-delete-item (menu item)
  "Deletes from MENU the item named ITEM.  x-menu-compute should be called
before the menu is made available to the user."
  (let ((old (current-buffer))
        elt)
    (set-buffer menu)
    (if (setq elt (assoc item x-menu-items-alist))
        (rplaca elt nil))
    (set-buffer old)
    item))

(defun x-menu-activate (menu)
  "Computes all necessary parameters for MENU.  This must be called whenever
a menu is modified before it is made available to the user.

This also creates the menu itself."
  (let ((buf (current-buffer)))
    (pop-to-buffer menu)
    (let (buffer-read-only)
      (setq x-menu-max-width (1- (screen-width)))
      (setq x-menu-item-width 0)
      (let (items-head
            (items-tail x-menu-items-alist))
        (while items-tail
          (if (caar items-tail)
              (progn (setq items-head (cons (car items-tail) items-head))
```

```
                    (setq x-menu-item-width
                          (max x-menu-item-width
                               (length (caar items-tail))))))
         (setq items-tail (cdr items-tail)))
       (setq x-menu-items-alist (reverse items-head)))
     (setq x-menu-item-width (+ x-menu-item-spacing x-menu-item-width))
     (setq x-menu-items-per-line
           (max 1 (/ x-menu-max-width x-menu-item-width)))
     (erase-buffer)
     (let ((items-head x-menu-items-alist))
       (while items-head
         (let ((items 0))
           (while (and items-head
                       (<= (setq items (1+ items)) x-menu-items-per-line))
             (insert (format (concat "%"
                                     (int-to-string x-menu-item-width) "s")
                             (caar items-head)))
             (setq items-head (cdr items-head))))
         (insert ?\n)))
     (shrink-window (max 0
                         (- (window-height)
                            (1+ (count-lines (point-min) (point-max))))))
     (goto-char (point-min)))
   (pop-to-buffer buf)))

(defun x-menu-pick-entry (position event)
  "Internal function for dispatching on mouse/menu events"
  (let* ((x (min (1- x-menu-items-per-line)
                 (/ (current-column) x-menu-item-width)))
         (y (- (count-lines (point-min) (point))
               (if (zerop (current-column)) 0 1)))
         (item (+ x (* y x-menu-items-per-line)))
         (litem (cdr (nth item x-menu-items-alist))))
    (and litem (funcall litem event)))
  (pop-to-buffer x-menu-assoc-buffer))
```

I.3. Scribe Mode

This example is the entire code for the Scribe mode package. Note its manipulation of the syntax table.

```
;; scribe mode, and its ideosyncratic commands.
;; Copyright (C) 1985 Free Software Foundation, Inc.

;; This file might become part of GNU Emacs.

;; GNU Emacs is distributed in the hope that it will be useful,
;; but without any warranty.  No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; document "GNU Emacs copying permission notice".   An exact copy
;; of the document is supposed to have been given to you along with
;; GNU Emacs so that you can know how you may redistribute it all.
;; It should be in a file named COPYING.  Among other things, the
;; copyright notice and this notice must be preserved on all copies.


(defvar scribe-mode-syntax-table nil
  "Syntax table used while in scribe mode.")

(defvar scribe-mode-abbrev-table nil
  "Abbrev table used while in scribe mode.")

(defvar scribe-fancy-paragraphs nil
  "*Non-NIL makes Scribe mode use a different style of paragraph
  separation.")

(defvar scribe-electric-quote nil
  "*Non-NIL makes insert of double quote use ‘‘ or ’’ depending on
  context.")

(defvar scribe-electric-parenthesis nil
  "*Non-NIL makes parenthesis char ( (]}> ) automatically insert its close
if typed after an @Command form.")

(defconst scribe-open-parentheses "[({<"
  "Open parenthesis characters for Scribe.")

(defconst scribe-close-parentheses "])}>"
  "Close parenthesis characters for Scribe.  These should match up with
scribe-open-parenthesis.")

(if (null scribe-mode-syntax-table)
    (let ((st (syntax-table)))
      (unwind-protect
       (progn
        (setq scribe-mode-syntax-table (copy-syntax-table
                                        text-mode-syntax-table))
        (set-syntax-table scribe-mode-syntax-table)
```

```
        (modify-syntax-entry ?\" "    ")
        (modify-syntax-entry ?\\ "    ")
        (modify-syntax-entry ?@ "w   ")
        (modify-syntax-entry ?< "(>  ")
        (modify-syntax-entry ?> ")<  ")
        (modify-syntax-entry ?[ "(]  ")
        (modify-syntax-entry ?] ")[  ")
        (modify-syntax-entry ?{ "(}  ")
        (modify-syntax-entry ?} "){  ")
        (modify-syntax-entry ?’ "w   "))
       (set-syntax-table st))))

(defvar scribe-mode-map nil)

(if scribe-mode-map
    nil
  (setq scribe-mode-map (make-sparse-keymap))
  (define-key scribe-mode-map "\t" ’scribe-tab)
  (define-key scribe-mode-map "\e\t" ’tab-to-tab-stop)
  (define-key scribe-mode-map "\es" ’center-line)
  (define-key scribe-mode-map "\e}" ’up-list)
  (define-key scribe-mode-map "\eS" ’center-paragraph)
  (define-key scribe-mode-map "\"" ’scribe-insert-quote)
  (define-key scribe-mode-map "(" ’scribe-parenthesis)
  (define-key scribe-mode-map "[" ’scribe-parenthesis)
  (define-key scribe-mode-map "{" ’scribe-parenthesis)
  (define-key scribe-mode-map "<" ’scribe-parenthesis)
  (define-key scribe-mode-map "\^cc" ’scribe-chapter)
  (define-key scribe-mode-map "\^cS" ’scribe-section)
  (define-key scribe-mode-map "\^cs" ’scribe-subsection)
  (define-key scribe-mode-map "\^ce" ’scribe-insert-environment)
  (define-key scribe-mode-map "\^c\^e" ’scribe-bracket-region-be)
  (define-key scribe-mode-map "\^c[" ’scribe-begin)
  (define-key scribe-mode-map "\^c]" ’scribe-end)
  (define-key scribe-mode-map "\^ci" ’scribe-italicize-word)
  (define-key scribe-mode-map "\^cb" ’scribe-bold-word)
  (define-key scribe-mode-map "\^cu" ’scribe-underline-word))

(defun scribe-mode ()
  "Major mode for editing files of Scribe (a text formatter) source.
Scribe-mode is similar text-mode, with a few extra commands added.
\\{scribe-mode-map}

Interesting variables:

scribe-electric-parenthesis
  Non-NIL makes Scribe mode use a different style of paragraph separation.

scribe-fancy-paragraphs
  Non-NIL makes insert of double quote use ‘‘ or ’’ depending on context.

scribe-electric-quote
  Non-NIL makes parenthesis char ( ([]}> ) automatically insert its close
if typed after an @Command form."
  (interactive)
  (kill-all-local-variables)
  (use-local-map scribe-mode-map)
```

```
  (setq mode-name "Scribe")
  (setq major-mode 'scribe-mode)
  (define-abbrev-table 'scribe-mode-abbrev-table ())
  (setq local-abbrev-table scribe-mode-abbrev-table)
  (make-local-variable 'comment-start)
  (setq comment-start "@Comment[")
  (make-local-variable 'comment-start-skip)
  (setq comment-start-skip (concat "@Comment[" scribe-open-parentheses "]"))
  (make-local-variable 'comment-column)
  (setq comment-column 0)
  (make-local-variable 'comment-end)
  (setq comment-end "]")
  (make-local-variable 'paragraph-start)
  (setq paragraph-start (concat "\\(^[\n\f]\\)\\|\\(^@\\w+["
                                 scribe-open-parentheses
                                "].*["
                                 scribe-close-parentheses
                                "]$\\)"))
  (make-local-variable 'paragraph-separate)
  (setq paragraph-separate (if scribe-fancy-paragraphs
                               paragraph-start "^$"))
  (make-local-variable 'compile-command)
  (setq compile-command (concat "scribe " (buffer-file-name)))
  (set-syntax-table scribe-mode-syntax-table)
  (run-hooks 'text-mode-hook 'scribe-mode-hook))

(defun scribe-tab ()
  (interactive)
  (insert "@\\"))

;; This algorithm could probably be improved somewhat.
;;  Right now, it loses seriously...

(defun scribe ()
  "Run Scribe on the current buffer."
  (interactive)
  (call-interactively 'compile))

(defun scribe-envelop-word (string count)
  "Surround current word with Scribe construct @STRING[...].  COUNT
specifies how many words to surround.  A negative count means to skip
backward."
  (let ((spos (point)) (epos (point)) (ccoun 0) noparens)
    (if (not (zerop count))
        (progn (if (= (char-syntax (preceding-char)) ?w)
                   (forward-sexp (min -1 count)))
               (setq spos (point))
               (if (looking-at (concat "@\\w[" scribe-open-parentheses "]"))
                   (forward-char 2)
                 (goto-char epos)
                 (skip-chars-backward "\\W")
                 (forward-char -1))
               (forward-sexp (max count 1))
               (setq epos (point))))
    (goto-char spos)
    (while (and (< ccoun (length scribe-open-parentheses))
                (save-excursion
```

```
                 (or (search-forward (char-to-string
                                       (aref scribe-open-parentheses ccoun))
                                      epos t)
                     (search-forward (char-to-string
                                       (aref scribe-close-parentheses
  ccoun))
                                      epos t)))
               (setq ccoun (1+ ccoun))))
    (if (>= ccoun (length scribe-open-parentheses))
        (progn (goto-char epos)
               (insert "@end(" string ")")
               (goto-char spos)
               (insert "@begin(" string ")"))
      (goto-char epos)
      (insert (aref scribe-close-parentheses ccoun))
      (goto-char spos)
      (insert "@" string (aref scribe-open-parentheses ccoun))
      (goto-char epos)
      (forward-char 3)
      (skip-chars-forward scribe-close-parentheses))))

(defun scribe-underline-word (count)
  "Underline COUNT words around point by means of Scribe constructs."
  (interactive "p")
  (scribe-envelop-word "u" count))

(defun scribe-bold-word (count)
  "Boldface COUNT words around point by means of Scribe constructs."
  (interactive "p")
  (scribe-envelop-word "b" count))

(defun scribe-italicize-word (count)
  "Italicize COUNT words around point by means of Scribe constructs."
  (interactive "p")
  (scribe-envelop-word "i" count))

(defun scribe-begin ()
  (interactive)
  (insert "\n")
  (forward-char -1)
  (scribe-envelop-word "Begin" 0)
  (re-search-forward (concat "[" scribe-open-parentheses "]")))

(defun scribe-end ()
  (interactive)
  (insert "\n")
  (forward-char -1)
  (scribe-envelop-word "End" 0)
  (re-search-forward (concat "[" scribe-open-parentheses "]")))

(defun scribe-chapter ()
  (interactive)
  (insert "\n")
  (forward-char -1)
  (scribe-envelop-word "Chapter" 0)
  (re-search-forward (concat "[" scribe-open-parentheses "]")))

(defun scribe-section ()
```

```
  (interactive)
  (insert "\n")
  (forward-char -1)
  (scribe-envelop-word "Section" 0)
  (re-search-forward (concat "[" scribe-open-parentheses "]")))

(defun scribe-subsection ()
  (interactive)
  (insert "\n")
  (forward-char -1)
  (scribe-envelop-word "SubSection" 0)
  (re-search-forward (concat "[" scribe-open-parentheses "]")))

(defun scribe-bracket-region-be (env min max)
  (interactive "sEnvironment: \nr")
  (save-excursion
    (goto-char max)
    (insert "@end(" env ")\n")
    (goto-char min)
    (insert "@begin(" env ")\n")))

(defun scribe-insert-environment (env)
  (interactive "sEnvironment: ")
  (scribe-bracket-region-be env (point) (point))
  (forward-line 1)
  (insert ?\n)
  (forward-char -1))

(defun scribe-insert-quote (count)
  "If scribe-electric-quote is non-NIL, insert ‘‘, ’’ or \" according
to preceding character.  With numeric arg N, always insert N \" characters.
Else just insert \"."
  (interactive "P")
  (if (or count (not scribe-electric-quote))
      (self-insert-command (prefix-numeric-value count))
    (let (lastfore lastback lastquote)
      (insert
       (cond
        ((= (preceding-char) ?\\) ?\")
        ((bobp) "‘‘")
        (t
         (setq lastfore (save-excursion (and (search-backward
                                              "‘‘" (- (point) 1000) t)
                                             (point)))
               lastback (save-excursion (and (search-backward
                                              "’’" (- (point) 1000) t)
                                             (point)))
               lastquote (save-excursion (and (search-backward
                                               "\"" (- (point) 100) t)
                                              (point))))
         (if (not lastquote)
             (cond ((not lastfore) "‘‘")
                   ((not lastback) "’’")
                   ((> lastfore lastback) "’’")
                   (t "‘‘"))
           (cond ((and (not lastback) (not lastfore)) "\"")
                 ((and lastback (not lastfore) (> lastquote lastback)) "\"")
```

```
                ((and lastback (not lastfore) (> lastback lastquote)) "‘‘")
                ((and lastfore (not lastback) (> lastquote lastfore)) "\"")
                ((and lastfore (not lastback) (> lastfore lastquote)) "’’")
                ((and (> lastquote lastfore) (> lastquote lastback)) "\"")
                ((> lastfore lastback) "’’")
                (t "‘‘")))))))))

(defun scribe-parenthesis (count)
  "If scribe-electric-parenthesis is non-NIL, insertion of an open-
  parenthesis
character inserts the following close parenthesis character if the
preceding text is of the form @Command."
  (interactive "P")
  (self-insert-command (prefix-numeric-value count))
  (let (at-command paren-char point-save)
    (if (or count (not scribe-electric-parenthesis))
        nil
      (save-excursion
        (forward-char -1)
        (setq point-save (point))
        (skip-chars-backward (concat "^ \n\t\f" scribe-open-parentheses))
        (setq at-command (and (equal (following-char) ?@)
                              (/= (point) (1- point-save)))))
      (if (and at-command
               (setq paren-char
                     (string-match (regexp-quote
                                    (char-to-string (preceding-char)))
                                   scribe-open-parentheses)))
          (save-excursion
            (insert (aref scribe-close-parentheses paren-char)))))))
```

I.4. Process Control

Some examples of process manipulation; the emacs server and display-time.

```
;; Lisp code for GNU Emacs running as server process.
;; Copyright (C) 1986 Free Software Foundation, Inc.
;; Author William Sommerfeld, wesommer@athena.mit.edu.

;; This file is part of GNU Emacs.

;; GNU Emacs is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY.  No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing.  Refer to the GNU Emacs General Public
;; License for full details.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; GNU Emacs General Public License.   A copy of this license is
;; supposed to have been given to you along with GNU Emacs so you
;; can know your rights and responsibilities.  It should be in a
;; file named COPYING.  Among other things, the copyright notice
;; and this notice must be preserved on all copies.
```

```
;;; This Lisp code is run in Emacs when it is to operate as
;;; a server for other processes.

;;; Load this library and do M-x server-start to enable Emacs as a server.
;;; Emacs runs the program ../etc/emacs-server as a subprocess
;;; for communication with clients.

;;; When some other program runs "the editor" to edit a file,
;;; "the editor" can be the Emacs client program ../etc/emacsclient.
;;; This program transmits the file names to Emacs through
;;; the server subprocess, and Emacs visits them and lets you edit them.

;;; When you are done editing and want "the editor" to return to the
;;; program that invoked it, do M-x server-done.  This uses
;;; the server communication subprocess to tell the client program
;;; to exit.

;;; Your editing commands and Emacs's display output go to and from
;;; the terminal in the usual way.  Thus, server operation is possible
;;; only when Emacs can talk to the terminal at the time you invoke
;;; the client.  This is possible in two cases:

;;; 1. On a window system, where Emacs runs in one window and the
;;; program that wants to use "the editor" runs in another.

;;; 2. When the program that wants to use "the editor" is running
;;; as a subprocess of Emacs.


;;; Start a server process running:

(defvar server-program
  "server"
  "*The program to use as the edit server")

(defvar server-process nil)

(defvar server-edit-buffers nil)

(defvar old-server-edit-buffer nil)

(defun server-start ()
  "Allow this Emacs process to be a server for client processes.
This starts a server communications subprocess through which
client \"editors\" can send your editing commands to this Emacs job."
  (interactive)
  (if server-process
      (if (or (not (eq (process-status server-process) 'run))
              (yes-or-no-p "A server process is running; kill it? "))
          (condition-case ()
                          (delete-process server-process)
            (error nil))
        (error "Cannot have two server processes")))
  (setq server-process nil)
  (condition-case ()
      (delete-file "~/.emacs_server")
    (error nil))
```

```
  (setq server-process
        (start-process "server" "*server*" server-program))
  (set-process-sentinel server-process 'server-sentinel))

(defun server-sentinel (proc msg)
  (cond ((eq (process-status proc) 'stop)
         (setq server-old-buffer (current-buffer))
         (setq server-edit-buffers nil)
         (switch-to-buffer "*server*")
         (goto-char (point-max))
         (forward-line -1)
         (if (looking-at "file: ")
             (progn
               (goto-char (match-end 0))
               (while (progn (switch-to-buffer "*server*")
                             (not (looking-at " *\n")))
                 (skip-chars-forward " ")
                 (setq start-filename (point))
                 (search-forward " " nil t)
                 (setq filename (buffer-substring start-filename (1-
  (point))))
                 (bury-buffer (current-buffer))
                 (find-file filename)
                 (setq server-edit-buffers (cons (current-buffer)
                                                 server-edit-buffers))
;                (set-minor-mode 'server-sentinel "Server" t)
; This is probably a bug: it will clobber some random major mode's
; local keymap!
;                (local-set-key "\^C\^C" 'server-done)
                 )
               (bury-buffer "*server*")
               (switch-to-buffer (car server-edit-buffers))
               (setq old-server-edit-buffer (car server-edit-buffers)))))
        (t (save-window-excursion
             (switch-to-buffer old-server-edit-buffer)
             (setq old-server-edit-buffer nil)
;            (set-minor-mode 'server-sentinel "Server" nil)
             ))))

(defun server-done ()
  "Say you are finished with server editing.
This saves the files that you are editing on the behest of the client
and causes the client to exit to the program that invoked it."
  (interactive)
  (if (not (memq (current-buffer) server-edit-buffers))
      (error "Not a current server edit buffer"))
  (while (car server-edit-buffers)
    (if (and (buffer-modified-p (car server-edit-buffers))
             (progn (switch-to-buffer (car server-edit-buffers))
                    (y-or-n-p (concat "Save file"
                                      (buffer-file-name
                                       (car server-edit-buffers))
                                      "? "))))
        (save-buffer (car server-edit-buffers)))
    (bury-buffer (current-buffer))
    (setq server-edit-buffers (cdr server-edit-buffers)))
  (switch-to-buffer server-old-buffer)
```

```
  (continue-process server-process))
```

And here is the server itself:

```
/* Communication subprocess for GNU Emacs acting as server.
   Copyright (C) 1986 Free Software Foundation, Inc.

This file is part of GNU Emacs.

GNU Emacs is distributed in the hope that it will be useful,
but without any warranty.  No author or distributor
accepts responsibility to anyone for the consequences of using it
or for whether it serves any particular purpose or works at all,
unless he says so in writing.

Everyone is granted permission to copy, modify and redistribute
GNU Emacs, but only under the conditions described in the
document "GNU Emacs copying permission notice".   An exact copy
of the document is supposed to have been given to you along with
GNU Emacs so that you can know how you may redistribute it all.
It should be in a file named COPYING.  Among other things, the
copyright notice and this notice must be preserved on all copies.  */


#ifndef lint
static char *rcsid_emacs_server_c = "$Header: emacs-server.c,v 1.2 86/10/24
  22:10:03 rlk Exp $";
#endif lint

/* The GNU Emacs edit server process is run as a subprocess of Emacs
   under control of the file lisp/server.el.
   This program accepts communication from client (program emacsclient.c)
   and passes their commands (consisting of keyboard characters)
   up to the Emacs which then executes them.  */

#include "../src/config.h"

#ifndef BSD
#include <stdio.h>

main ()
{
  fprintf (stderr, "Sorry, the Emacs server is supported only on Berkeley
  Unix.\n");
  exit (1);
}

#else /* BSD */

#include <sys/types.h>
#include <sys/socket.h>
#include <sys/signal.h>
#include <sys/un.h>
/* "THIS has to be fixed.  Remember, stderr may not exist...-rlk."
   Incorrect.  This program runs as an inferior of Emacs.
   Its stderr always exists--rms.  */
```

```
#include <stdio.h>

main ()
{
  int s, infd, fromlen;
  struct sockaddr_un server, fromunix;
  char *homedir, *getenv ();
  char string[BUFSIZ];
  FILE *infile;

  /*
   * Open up an AF_UNIX socket in this person’s home directory
   */

  if ((s = socket (AF_UNIX, SOCK_STREAM, 0)) < 0)
    {
      perror ("socket");
      exit (1);
    }
  server.sun_family = AF_UNIX;
  if ((homedir = getenv ("HOME")) == NULL)
    {
      fprintf (stderr,"No home directory\n");
      exit (1);
    }
  strcpy (server.sun_path, homedir);
  strcat (server.sun_path, "/.emacs_server");
  if (bind (s, &server, strlen (server.sun_path) + 2) < 0)
    {
      perror ("bind");
      exit (1);
    }
  /*
   * Now, just wait for everything to come in..
   */
  if (listen (s, 5) < 0)
    {
      perror ("listen");
      exit (1);
    }

  /* Disable sigpipes in case luser kills client... */
  signal (SIGPIPE, SIG_IGN);
  for (;;)
    {
      fromlen = sizeof (fromunix);
      fromunix.sun_family = AF_UNIX;
      if ((infd = accept (s, &fromunix, &fromlen)) < 0)
        {
          perror ("accept");
          close (infd);
          continue;
        }
      else if ((infile = fdopen (infd, "r+")) == NULL)
        {
          perror ("fdopen");
          close (infd);         /* Prevent descriptor leak.. */
```

```
            continue;
          }
        else if (fgets (string, BUFSIZ, infile) == NULL)
          {
            perror ("fgets");
          }
        else
          {
            printf ("file: %s", string); /* Newline is part of string.. */
            fflush (stdout);
            /* Now, wait for a wakeup */
            kill (getpid (), SIGSTOP);
            fprintf (infile, "done\n");
            fflush (infile);
          }
        fclose (infile);
    }
}

#endif /* BSD */
```

This displays the time and load average in the mode line of GNU Emacs. This is the code from Version 18, which has enhanced mode line facilities; the process control is identical in Version 17.

```
;; Display time and load in mode line of Emacs.
;; Copyright (C) 1985, 1986 Free Software Foundation, Inc.

;; This file is part of GNU Emacs.

;; GNU Emacs is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY.  No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing.  Refer to the GNU Emacs General Public
;; License for full details.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; GNU Emacs General Public License.   A copy of this license is
;; supposed to have been given to you along with GNU Emacs so you
;; can know your rights and responsibilities.  It should be in a
;; file named COPYING.  Among other things, the copyright notice
;; and this notice must be preserved on all copies.


(defvar display-time-process nil)

(defvar display-time-interval 60
  "*Seconds between updates of time in the mode line.")

(defvar display-time-string nil)

(defun display-time ()
  "Display current time and load level in mode line of each buffer.
Updates automatically every minute.
If display-time-day-and-date is non-nil, the current day and date
are displayed as well."
```

```
  (interactive)
  (let ((live (and display-time-process
                   (eq (process-status display-time-process) 'run))))
    (if (not live)
        (save-excursion
          (if display-time-process
              (delete-process display-time-process))
          (or global-mode-string (setq global-mode-string '("")))
          (or (memq 'display-time-string global-mode-string)
              (setq global-mode-string
                    (append global-mode-string '(display-time-string))))
          (setq display-time-string "time and load")
          (setq display-time-process
                (start-process "display-time" nil
                               "loadst"
                               "-n" (int-to-string display-time-interval)))
          (process-kill-without-query display-time-process)
          (set-process-sentinel display-time-process 'display-time-sentinel)
          (set-process-filter display-time-process 'display-time-filter)))))

(defun display-time-sentinel (proc reason)
  (or (eq (process-status proc) 'run)
      (setq display-time-string ""))
  ;; Force mode-line updates
  (save-excursion (set-buffer (other-buffer)))
  (set-buffer-modified-p (buffer-modified-p))
  (sit-for 0))

(defun display-time-filter (proc string)
  ;; Desired data can't need more than the last 30 chars,
  ;; so save time by flushing the rest.
  ;; This way, if we have many different times all collected at once,
  ;; we can discard all but the last few very fast.
  (if (> (length string) 30)
      (setq string (substring string -30)))
  ;; Now discard all but the very last one.
  (while (string-match "[0-9]+:" string 4)
    (setq string (substring string (match-beginning 0))))
  ;; Append the date if desired.
  (if display-time-day-and-date
      (setq string (concat (substring (current-time-string) 0 11) string)))
  ;; Install the new time for display.
  (setq display-time-string string)
  ;; Force redisplay of all buffers' mode lines to be considered.
  (save-excursion (set-buffer (other-buffer)))
  (set-buffer-modified-p (buffer-modified-p))
  ;; Do redisplay right now, if no input pending.
  (sit-for 0))
```

And finally, shell mode, one of the most complex examples of process manipulation in the current distribution of emacs. If you want to write code that hacks processes, you can probably find something useful in here.

```
;; Run subshell under Emacs
;; Copyright (C) 1985, 1986 Free Software Foundation, Inc.
```

```
;; This file is part of GNU Emacs.

;; GNU Emacs is distributed in the hope that it will be useful,
;; but WITHOUT ANY WARRANTY.  No author or distributor
;; accepts responsibility to anyone for the consequences of using it
;; or for whether it serves any particular purpose or works at all,
;; unless he says so in writing.  Refer to the GNU Emacs General Public
;; License for full details.

;; Everyone is granted permission to copy, modify and redistribute
;; GNU Emacs, but only under the conditions described in the
;; GNU Emacs General Public License.   A copy of this license is
;; supposed to have been given to you along with GNU Emacs so you
;; can know your rights and responsibilities.  It should be in a
;; file named COPYING.  Among other things, the copyright notice
;; and this notice must be preserved on all copies.


(provide ’shell)

;; For old Emacs versions.
(or (fboundp ’process-send-string)
    (progn
      (fset ’process-send-string ’send-string)
      (fset ’process-send-region ’send-region)))

(defvar last-input-start nil
  "In a shell-mode buffer, marker for start of last unit of input.")
(defvar last-input-end nil
  "In a shell-mode buffer, marker for start of last unit of input.")

(defvar shell-mode-map nil)

(defvar shell-directory-stack nil
  "List of directories saved by pushd in this buffer’s shell.")

(defvar shell-popd-regexp "popd"
  "*Regexp to match subshell commands equivalent to popd.")

(defvar shell-pushd-regexp "pushd"
  "*Regexp to match subshell commands equivalent to pushd.")

(defvar shell-cd-regexp "cd"
  "*Regexp to match subshell commands equivalent to cd.")

(defvar explicit-shell-file-name nil
  "*If non-nil, is file name to use for explicitly requested inferior
  shell.")

;In loaddefs.el now.
;(defconst shell-prompt-pattern
;  "^[^#$%>]*[#$%>] *"
;  "*Regexp used by Newline command to match subshell prompts.
;Anything from beginning of line up to the end of what this pattern matches
;is deemed to be prompt, and is not reexecuted.")

(defun shell-mode ()
```

```
  "Major mode for interacting with an inferior shell.
Shell name is same as buffer name, sans the asterisks.
Return at end of buffer sends line as input.
Return not at end copies rest of line to end and sends it.

The following commands imitate the usual Unix interrupt and
editing control characters:
\\{shell-mode-map}

Entry to this mode calls the value of shell-mode-hook with no args,
if that value is non-nil.

cd, pushd and popd commands given to the shell are watched
by Emacs to keep this buffer’s default directory
the same as the shell’s working directory.
Variables shell-cd-regexp, shell-pushd-regexp and shell-popd-regexp
are used to match these command names.

You can send text to the shell (or its subjobs) from other buffers
using the commands process-send-region, process-send-string
and lisp-send-defun."
  (interactive)
  (kill-all-local-variables)
  (setq major-mode ’shell-mode)
  (setq mode-name "Shell")
  (setq mode-line-process ’(": %s"))
  (use-local-map shell-mode-map)
  (make-local-variable ’shell-directory-stack)
  (setq shell-directory-stack nil)
  (make-local-variable ’last-input-start)
  (setq last-input-start (make-marker))
  (make-local-variable ’last-input-end)
  (setq last-input-end (make-marker))
  (run-hooks ’shell-mode-hook))

(if shell-mode-map
    nil
  (setq shell-mode-map (make-sparse-keymap))
  (define-key shell-mode-map "\C-m" ’shell-send-input)
  (define-key shell-mode-map "\C-c\C-d" ’shell-send-eof)
  (define-key shell-mode-map "\C-c\C-u" ’kill-shell-input)
  (define-key shell-mode-map "\C-c\C-w" ’backward-kill-word)
  (define-key shell-mode-map "\C-c\C-c" ’interrupt-shell-subjob)
  (define-key shell-mode-map "\C-c\C-z" ’stop-shell-subjob)
  (define-key shell-mode-map "\C-c\C-\\" ’quit-shell-subjob)
  (define-key shell-mode-map "\C-c\C-o" ’kill-output-from-shell)
  (define-key shell-mode-map "\C-c\C-r" ’show-output-from-shell)
  (define-key shell-mode-map "\C-c\C-y" ’copy-last-shell-input))

(defun shell ()
  "Run an inferior shell, with I/O through buffer *shell*.
If buffer exists but shell process is not running, make new shell.
Program used comes from variable explicit-shell-file-name,
 or (if that is nil) from the ESHELL environment variable,
 or else from SHELL if there is no ESHELL.
If a file ~/.emacs_SHELLNAME exists, it is given as initial input
 (Note that this may lose due to a timing error if the shell
```

```
  discards input when it starts up.)
The buffer is put in shell-mode, giving commands for sending input
and controlling the subjobs of the shell.  See shell-mode.
See also variable shell-prompt-pattern.

Note that many people's .cshrc files unconditionally clear the prompt.
If yours does, you will probably want to change it."
  (interactive)
  (let* ((prog (or explicit-shell-file-name
                   (getenv "ESHELL")
                   (if (eq system-type 'hpux) "sh"
                     ;; On hpux people normally use csh,
                     ;; but the csh in hpux has stty sanity checking
                     ;; so it does not work under emacs.
                     (getenv "SHELL"))
                   "/bin/sh"))
         (name (file-name-nondirectory prog)))
    (switch-to-buffer
     (make-shell "shell" prog
                 (if (file-exists-p (concat "~/.emacs_" name))
                     (concat "~/.emacs_" name))
                 "-i"))))

(defun make-shell (name program &optional startfile &rest switches)
  (let ((buffer (get-buffer-create (concat "*" name "*")))
        proc status size)
    (setq proc (get-buffer-process buffer))
    (if proc
        (setq status (process-status proc)))
    (save-excursion
      (set-buffer buffer)
      ;;    (setq size (buffer-size))
      (if (memq status '(run stop))
          nil
        (if proc (delete-process proc))
        (setq proc (apply 'start-process (append (list name buffer program)
  switches)))
        (cond (startfile
               ;;This is guaranteed to wait long enough
               ;;but has bad results if the shell does not prompt at all
               ;;            (while (= size (buffer-size))
               ;;              (sleep-for 1))
               ;;I hope 1 second is enough!
               (sleep-for 1)
               (goto-char (point-max))
               (insert-file-contents startfile)
               (setq startfile (buffer-substring (point) (point-max)))
               (delete-region (point) (point-max))
               (process-send-string proc startfile)))
        (setq name (process-name proc)))
      (goto-char (point-max))
      (set-marker (process-mark proc) (point))
      (shell-mode))
    buffer))

(defvar shell-set-directory-error-hook 'ignore
  "Function called with no arguments when shell-send-input
```

```
recognizes a change-directory command but gets an error
trying to change Emacs's default directory.")

(defun shell-send-input ()
  "Send input to subshell.
At end of buffer, sends all text after last output
 as input to the subshell, including a newline inserted at the end.
Not at end, copies current line to the end of the buffer and sends it,
after first attempting to discard any prompt at the beginning of the line
by matching the regexp that is the value of shell-prompt-pattern if
  possible.
This regexp should start with \"^\"."
  (interactive)
  (end-of-line)
    (if (eobp)
        (progn
          (move-marker last-input-start
                       (process-mark (get-buffer-process (current-buffer))))
          (insert ?\n)
          (move-marker last-input-end (point)))
    (beginning-of-line)
    (re-search-forward shell-prompt-pattern nil t)
    (let ((copy (buffer-substring (point)
                                  (progn (forward-line 1) (point)))))
      (goto-char (point-max))
      (move-marker last-input-start (point))
      (insert copy)
      (move-marker last-input-end (point))))
    ;; Even if we get an error trying to hack the working directory,
    ;; still send the input to the subshell.
    (condition-case ()
        (save-excursion
          (goto-char last-input-start)
          (shell-set-directory))
      (funcall shell-set-directory-error-hook))
  (let ((process (get-buffer-process (current-buffer))))
    (process-send-region process last-input-start last-input-end)
    (set-marker (process-mark process) (point))))

;;;  If this code changes (shell-send-input and shell-set-directory),
;;;  the customization tutorial in
;;;  info/customizing-tutorial must also change, since it explains this
;;;  code.  Please let marick@gswd-vms.arpa know of any changes you
;;;  make.

(defun shell-set-directory ()
  (cond ((and (looking-at shell-popd-regexp)
              (memq (char-after (match-end 0)) '(?\; ?\n)))
         (if shell-directory-stack
             (progn
               (cd (car shell-directory-stack))
               (setq shell-directory-stack (cdr shell-directory-stack)))))
        ((looking-at shell-pushd-regexp)
         (cond ((memq (char-after (match-end 0)) '(?\; ?\n))
                (if shell-directory-stack
                    (let ((old default-directory))
                      (cd (car shell-directory-stack))
```

```
                        (setq shell-directory-stack
                              (cons old (cdr shell-directory-stack))))))
                 ((memq (char-after (match-end 0)) '(?\  ?\t))
                  (let (dir)
                    (skip-chars-forward "^ ")
                    (skip-chars-forward " \t")
                    (if (file-directory-p
                          (setq dir
                                (expand-file-name
                                  (substitute-in-file-name
                                    (buffer-substring
                                      (point)
                                      (progn
                                        (skip-chars-forward "^\n \t;")
                                        (point)))))))
                        (progn
                          (setq shell-directory-stack
                                (cons default-directory shell-directory-
  stack))
                          (cd dir)))))))
          ((looking-at shell-cd-regexp)
           (cond ((memq (char-after (match-end 0)) '(?\; ?\n))
                  (cd (getenv "HOME")))
                 ((memq (char-after (match-end 0)) '(?\  ?\t))
                  (let (dir)
                    (forward-char 3)
                    (skip-chars-forward " \t")
                    (if (file-directory-p
                          (setq dir
                                (expand-file-name
                                  (substitute-in-file-name
                                    (buffer-substring
                                      (point)
                                      (progn
                                        (skip-chars-forward "^\n \t;")
                                        (point)))))))
                        (cd dir))))))))

(defun shell-send-eof ()
  "Send eof to subshell (or to the program running under it)."
  (interactive)
  (process-send-eof))

(defun kill-output-from-shell ()
  "Kill all output from shell since last input."
  (interactive)
  (goto-char (point-max))
  (kill-region last-input-end (point))
  (insert "> output flushed ***\n"))

(defun show-output-from-shell ()
  "Display start of this batch of shell output at top of window.
Also put cursor there."
  (interactive)
  (set-window-start (selected-window) last-input-end)
  (goto-char last-input-end))

(defun copy-last-shell-input ()
```

```
  "Copy previous shell input, sans newline, and insert before point."
  (interactive)
  (insert (buffer-substring last-input-end last-input-start))
  (delete-char -1))

(defun interrupt-shell-subjob ()
  "Interrupt this shell's current subjob."
  (interactive)
  (interrupt-process nil t))

(defun kill-shell-subjob ()
  "Send kill signal to this shell's current subjob."
  (interactive)
  (kill-process nil t))

(defun quit-shell-subjob ()
  "Send quit signal to this shell's current subjob."
  (interactive)
  (quit-process nil t))

(defun stop-shell-subjob ()
  "Stop this shell's current subjob."
  (interactive)
  (stop-process nil t))

(defun kill-shell-input ()
  "Kill all text since last stuff output by the shell or its subjobs."
  (interactive)
  (kill-region (process-mark (get-buffer-process (current-buffer)))
               (point)))

(defvar inferior-lisp-mode-map nil)
(if inferior-lisp-mode-map
    nil
  (setq inferior-lisp-mode-map (copy-alist shell-mode-map))
  (lisp-mode-commands inferior-lisp-mode-map)
  (define-key inferior-lisp-mode-map "\e\C-x" 'lisp-send-defun))

(defun inferior-lisp-mode ()
  "Major mode for interacting with an inferior Lisp process.

The following commands are available:
\\{inferior-lisp-mode-map}

Entry to this mode calls the value of lisp-mode-hook with no arguments,
if that value is non-nil.  Likewise with the value of shell-mode-hook.
lisp-mode-hook is called after shell-mode-hook.

You can send text to the inferior Lisp from other buffers
using the commands process-send-region, process-send-string
and \\[lisp-send-defun].

Commands:
Delete converts tabs to spaces as it moves back.
Tab indents for Lisp; with argument, shifts rest
 of expression rigidly with the current line.
Meta-Control-Q does Tab on each line starting within following expression.
```

```
Paragraphs are separated only by blank lines.  Semicolons start comments.

Return at end of buffer sends line as input.
Return not at end copies rest of line to end and sends it.
C-d at end of buffer sends end-of-file as input.
C-d not at end or with arg deletes or kills characters.
C-u and C-w are kill commands, imitating normal Unix input editing.
C-c interrupts the shell or its current subjob if any.
C-z stops, likewise.  C-\\ sends quit signal, likewise.

C-x C-k deletes last batch of output from shell.
C-x C-v puts top of last batch of output at top of window."
  (interactive)
  (kill-all-local-variables)
  (setq major-mode 'inferior-lisp-mode)
  (setq mode-name "Inferior Lisp")
  (setq mode-line-process '(": %s"))
  (lisp-mode-variables)
  (use-local-map inferior-lisp-mode-map)
  (make-local-variable 'last-input-start)
  (setq last-input-start (make-marker))
  (make-local-variable 'last-input-end)
  (setq last-input-end (make-marker))
  (run-hooks 'shell-mode-hook 'lisp-mode-hook))

(defun run-lisp ()
  "Run an inferior Lisp process, input and output via buffer *lisp*."
  (interactive)
  (switch-to-buffer (make-shell "lisp" "lisp"))
  (inferior-lisp-mode))

(defun lisp-send-defun ()
  "Send the current defun to the Lisp process made by M-x run-lisp."
  (interactive)
  (save-excursion
   (end-of-defun)
   (let ((end (point)))
     (beginning-of-defun)
     (process-send-region "lisp" (point) end)
     (process-send-string "lisp" "\n"))))

(defun lisp-send-defun-and-go ()
  "Send the current defun to the inferior Lisp, and switch to *lisp*
  buffer."
  (interactive)
  (lisp-send-defun)
  (switch-to-buffer "*lisp*"))
```

I.5. Some Miscellaneous Examples

This code sample contrasts the definition of one of the most complex major modes in the current system with one of the simplest. If you want to look at the entire package, you'll have to look at the source code.

```
;;; Rmail mode -- one of the most complicated major modes running in
;;; the system.

(if rmail-mode-map
    nil
  (setq rmail-mode-map (make-keymap))
  (suppress-keymap rmail-mode-map)
  (define-key rmail-mode-map "." 'rmail-beginning-of-message)
  (define-key rmail-mode-map " " 'scroll-up)
  (define-key rmail-mode-map "\177" 'scroll-down)
  (define-key rmail-mode-map "n" 'rmail-next-undeleted-message)
  (define-key rmail-mode-map "p" 'rmail-previous-undeleted-message)
  (define-key rmail-mode-map "\en" 'rmail-next-message)
  (define-key rmail-mode-map "\ep" 'rmail-previous-message)
  (define-key rmail-mode-map "\e\C-n" 'rmail-next-labeled-message)
  (define-key rmail-mode-map "\e\C-p" 'rmail-previous-labeled-message)
  (define-key rmail-mode-map "\C-xn" 'rmail-next-unseen-message)
  (define-key rmail-mode-map "\C-xp" 'rmail-previous-unseen-message)
  (define-key rmail-mode-map "\C-x\C-n" 'rmail-next-default-labeled-message)
  (define-key rmail-mode-map "\C-x\C-p" 'rmail-previous-default-labeled-
  message)
  (define-key rmail-mode-map "a" 'rmail-add-label)
  (define-key rmail-mode-map "k" 'rmail-kill-label)
  (define-key rmail-mode-map "d" 'rmail-delete-forward)
  (define-key rmail-mode-map "u" 'rmail-undelete-previous-message)
  (define-key rmail-mode-map "\eu" 'rmail-undelete-message)
  (define-key rmail-mode-map "e" 'rmail-expunge)
  (define-key rmail-mode-map "s" 'rmail-save)
  (define-key rmail-mode-map "g" 'rmail-get-new-mail)
  (define-key rmail-mode-map "h" 'rmail-summary)
  (define-key rmail-mode-map "\e\C-h" 'rmail-summary)
  (define-key rmail-mode-map "l" 'rmail-summary-by-labels)
  (define-key rmail-mode-map "\e\C-l" 'rmail-summary-by-labels)
  (define-key rmail-mode-map "\e\C-r" 'rmail-summary-by-recipients)
  (define-key rmail-mode-map "t" 'rmail-toggle-header)
  (define-key rmail-mode-map "m" 'rmail-mail)
  (define-key rmail-mode-map "r" 'rmail-reply)
  (define-key rmail-mode-map "c" 'rmail-continue)
  (define-key rmail-mode-map "f" 'rmail-forward)
  (define-key rmail-mode-map "\es" 'rmail-search)
  (define-key rmail-mode-map "j" 'rmail-show-message)
  (define-key rmail-mode-map "o" 'rmail-output-to-rmail-file)
  (define-key rmail-mode-map "\C-o" 'rmail-output)
  (define-key rmail-mode-map "i" 'rmail-input)
  (define-key rmail-mode-map "q" 'rmail-quit)
  (define-key rmail-mode-map ">" 'rmail-last-message)
  (define-key rmail-mode-map "?" 'describe-mode)
  (define-key rmail-mode-map "\C-r" 'rmail-edit-current-message)
  (define-key rmail-mode-map "\C-d" 'rmail-delete-backward))

(defun rmail-mode ()
```

```
  "Rmail Mode is used by \\[rmail] for editing Rmail files.
All normal editing commands are turned off.
Instead, these commands are available:

.       Move point to front of this message (same as \\[beginning-of-
  buffer]).
SPC     Scroll to next screen of this message.
DEL     Scroll to previous screen of this message.
n       Move to Next non-deleted message.
p       Move to Previous non-deleted message.
M-n     Move to Next message whether deleted or not.
M-p     Move to Previous message whether deleted or not.
>       Move to the last message in Rmail file.
j       Jump to message specified by numeric position in file.
M-s     Search for string and show message it is found in.
d       Delete this message, move to next nondeleted.
C-d     Delete this message, move to previous nondeleted.
u       Undelete previous message.  M-u undelete this message.
e       Expunge deleted messages.
s       Save the file (same as C-x C-s).
g       Move new mail from system spool directory or mbox into this file.
m       Mail a message (same as \\[mail-other-window]).
c       Continue composing outgoing message started before.
r       Reply to this message.  Like m but initializes some fields.
f       Forward this message to another user.
o       Output this message to an Rmail file (append it).
C-o     Output this message to a Unix-format mail file (append it).
i       Input Rmail file.  Run Rmail on that file.
q       Quit out of Rmail and save Rmail file.
a       Add label to message.  It will be displayed in the mode line.
k       Kill label.  Remove a label from current message.
C-M-n   Move to Next message with specified label
          (label defaults to last one specified).
          Standard labels: filed, unseen, answered, forwarded, deleted.
          Any other label is present only if you add it with ‘a’.
C-M-p   Move to Previous message with specified label
C-x n   Move to Next unseen message.
C-x p   Move to Previous unseen message.
C-x C-n Move to Next message with default (last used) label.
C-x C-p Move to Previous message with default label.
C-M-h   Show headers buffer, with a one line summary of each message.
C-M-l   Like h only just messages with particular label(s) are summarized.
C-M-r   Like h only just messages with particular recipient(s) are
  summarized.
t       Toggle header, show Rmail header if unformatted or vice versa.
C-r     Edit the current message.  C-c C-c to return to Rmail."
  (interactive)
  (kill-all-local-variables)
  (rmail-mode-1)
  (rmail-variables)
  (run-hooks ’rmail-mode-hook))         ;Thank you rms!

(defun rmail-mode-1 ()
  (setq major-mode ’rmail-mode)
  (setq mode-name "RMAIL")
  (setq buffer-read-only t)
  ;; No need to auto save RMAIL files.
```

```
  (setq buffer-auto-save-file-name nil)
  (setq mode-line-format "--- Emacs: %b  %M  %[(%m)%] ----%3p-%-")
  (use-local-map rmail-mode-map)
  (set-syntax-table text-mode-syntax-table)
  (setq local-abbrev-table text-mode-abbrev-table))

(defun rmail-variables ()
  (make-local-variable 'rmail-last-label)
  (make-local-variable 'rmail-deleted-vector)
  (make-local-variable 'rmail-keywords)
  (make-local-variable 'rmail-summary-buffer)
  (make-local-variable 'rmail-summary-vector)
  (make-local-variable 'rmail-current-message)
  (make-local-variable 'rmail-total-messages)
  (make-local-variable 'require-final-newline)
  (setq require-final-newline nil)
  (make-local-variable 'version-control)
  (setq version-control 'never)
  (make-local-variable 'rmail-message-vector)
  (make-local-variable 'rmail-last-file)
  (make-local-variable 'rmail-inbox-list)
  (setq rmail-inbox-list (rmail-parse-file-inboxes)))

;;;  Rmail-edit-mode -- a very simple major mode.  Note that it
;;;  essentially is text mode with a few additions.
(defvar rmail-edit-map (copy-keymap text-mode-map))

(define-key rmail-edit-map "\C-c\C-c" 'rmail-cease-edit)
(define-key rmail-edit-map "\C-c\C-]" 'rmail-abort-edit)
(define-key rmail-edit-map "\C-c\C-\\" 'mail-fill-yanked-message)

(defun rmail-edit-mode ()
  "Major mode for editing the contents of an RMAIL message.
The editing commands are the same as in Text mode, together with two
  commands
to return to regular RMAIL:
  *  rmail-abort-edit (\\[rmail-abort-edit]) cancels the changes
     you have made and returns to RMAIL
  *  rmail-cease-edit (\\[rmail-cease-edit]) makes them permanent.
\\{rmail-edit-map}"
  (use-local-map rmail-edit-map)
  (setq major-mode 'rmail-edit-mode)
  (setq mode-name "RMAIL Edit")
  (setq mode-line-format default-mode-line-format)
  (run-hooks 'text-mode-hook 'rmail-edit-mode-hook))

;;;  Fundamental mode -- the simplest of them all

(defun fundamental-mode ()
  "Major mode not specialized for anything in particular.
Other major modes are defined by comparison with this one."
  (interactive)
  (kill-all-local-variables))

;;; Some miscellaneous functions from simple.el

(defun what-cursor-position ()
```

```
  "Print info on cursor position (on screen and within buffer)."
  (interactive)
  (let* ((char (following-char))
         (beg (point-min))
         (end (point-max))
         (pos (point))
         (total (buffer-size))
         (percent (if (> total 50000)
                      ;; Avoid overflow from multiplying by 100!
                      (/ (+ (/ total 200) (1- pos)) (max (/ total 100) 1))
                    (/ (+ (/ total 2) (* 100 (1- pos))) (max total 1))))
         (hscroll (if (= (window-hscroll) 0)
                      ""
                    (format " Hscroll=%d" (window-hscroll))))
         (col (current-column)))
    (if (= pos end)
        (if (or (/= beg 1) (/= end (1+ total)))
            (message "point=%d of %d(%d%%) <%d - %d>  x=%d %s"
                     pos total percent beg end col hscroll)
          (message "point=%d of %d(%d%%)  x=%d %s"
                   pos total percent col hscroll))
      (if (or (/= beg 1) (/= end (1+ total)))
          (message "Char: %s (0%o)  point=%d of %d(%d%%) <%d - %d>  x=%d %s"
                   (single-key-description char) char pos total percent beg
  end col hscroll)
        (message "Char: %s (0%o)  point=%d of %d(%d%%)  x=%d %s"
                 (single-key-description char) char pos total percent col
  hscroll)))))

;;; Even this is done in Lisp!  It isn't called a minor mode, but
;;; essentially is one.

(defun blink-matching-open ()
  "Move cursor momentarily to the beginning of the sexp before point."
  (and (> (point) (1+ (point-min)))
       (/= (char-syntax (char-after (- (point) 2))) ?\\ )
       blink-matching-paren
       (let* ((oldpos (point))
              (blinkpos)
              (mismatch))
         (save-excursion
           (save-restriction
             (if blink-matching-paren-distance
                 (narrow-to-region (max (point-min)
                                        (- (point) blink-matching-paren-
  distance))
                                   oldpos))
             (condition-case ()
                 (setq blinkpos (scan-sexps oldpos -1))
               (error nil)))
           (and blinkpos (/= (char-syntax (char-after blinkpos))
                             ?\$)
                (setq mismatch
                      (/= last-input-char
                          (logand (lsh (aref (syntax-table)
                                             (char-after blinkpos))
                                       -8)
```

```
                             ?\177))))
          (if mismatch (setq blinkpos nil))
          (if blinkpos
              (progn
               (goto-char blinkpos)
               (if (pos-visible-in-window-p)
                   (sit-for 1)
                 (goto-char blinkpos)
                 (message
                  "Matches %s"
                  (if (save-excursion
                        (skip-chars-backward " \t")
                        (not (bolp)))
                      (buffer-substring (progn (beginning-of-line) (point))
                                        (1+ blinkpos))
                    (buffer-substring blinkpos
                                      (progn
                                       (forward-char 1)
                                       (skip-chars-forward "\n \t")
                                       (end-of-line)
                                       (point)))))))
            (cond (mismatch
                   (message "Mismatched parentheses"))
                  ((not blink-matching-paren-distance)
                   (message "Unmatched parenthesis"))))))))

;;; And finally, a group of related commands.

(defun transpose-chars (arg)
  "Interchange characters around point, moving forward one character.
With prefix arg ARG, effect is to take character before point
and drag it forward past ARG other characters (backward if ARG negative).
If no argument and at end of line, the previous two chars are exchanged."
  (interactive "*P")
  (and (null arg) (eolp) (forward-char -1))
  (transpose-subr 'forward-char (prefix-numeric-value arg)))

(defun transpose-words (arg)
  "Interchange words around point, leaving point at end of them.
With prefix arg ARG, effect is to take word before or around point
and drag it forward past ARG other words (backward if ARG negative).
If ARG is zero, the words around or after point and around or after mark
are interchanged."
  (interactive "*p")
  (transpose-subr 'forward-word arg))

(defun transpose-sexps (arg)
  "Like \\[transpose-words] but applies to sexps.
Does not work on a sexp that point is in the middle of
if it is a list or string."
  (interactive "*p")
  (transpose-subr 'forward-sexp arg))

(defun transpose-lines (arg)
  "Exchange current line and previous line, leaving point after both.
With argument ARG, takes previous line and moves it past ARG lines.
With argument 0, interchanges line point is in with line mark is in."
```

```
  (interactive "*p")
  (transpose-subr 'forward-line arg))

(defun transpose-subr (mover arg)
  (let (start1 end1 start2 end2)
    (if (= arg 0)
        (progn
          (save-excursion
            (funcall mover 1)
            (setq end2 (point))
            (funcall mover -1)
            (setq start2 (point))
            (goto-char (mark))
            (funcall mover 1)
            (setq end1 (point))
            (funcall mover -1)
            (setq start1 (point))
            (transpose-subr-1))
          (exchange-point-and-mark)))
    (while (> arg 0)
      (funcall mover -1)
      (setq start1 (point))
      (funcall mover 1)
      (setq end1 (point))
      (funcall mover 1)
      (setq end2 (point))
      (funcall mover -1)
      (setq start2 (point))
      (transpose-subr-1)
      (goto-char end2)
      (setq arg (1- arg)))
    (while (< arg 0)
      (funcall mover -1)
      (setq start2 (point))
      (funcall mover -1)
      (setq start1 (point))
      (funcall mover 1)
      (setq end1 (point))
      (funcall mover 1)
      (setq end2 (point))
      (transpose-subr-1)
      (setq arg (1+ arg)))))

(defun transpose-subr-1 ()
  (let ((word1 (buffer-substring start1 end1))
        (word2 (buffer-substring start2 end2)))
    (delete-region start2 end2)
    (goto-char start2)
    (insert word1)
    (goto-char (if (< start1 start2) start1
                 (+ start1 (- (length word1) (length word2)))))
    (delete-char (length word1))
    (insert word2)))
```

Table of Contents