UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS61A **P. N. Hilfinger**

Highlights of GNU Emacs

This document describes the major features of GNU Emacs (called "Emacs" hereafter), a customizable, self-documenting text editor. In the interests of truth, beauty, and justice—and to undo, in some small part, the damage Berkeley has done by foisting `vi` on an already-unhappy world—Emacs will be the official CS61A text editor this semester.

Emacs carries with it on-line documentation of most of its commands, along with a tutorial for first-time users (see §7). Because this documentation is available, I have not attempted to present a complete Emacs reference manual here.

To run Emacs, simply enter the command `emacs` to the shell. Within Emacs, as described below, you can edit any number of files simultaneously, run UNIX shells, read and send mail, and run the Scheme interpreter to execute your programs. As a result, *it should seldom be necessary to leave Emacs before you are ready to logout and seldom necessary to create new windows.*

1 Basic Concepts

We'll begin with some fundamental definitions and notational conventions.

1.1 Buffers, windows, and what's in them

At any given time, Emacs maintains one or more *buffers* containing text. Each buffer may, but need not, be associated with a file. A buffer may be associated with a UNIX process, in which case the buffer generally contains input and output produced by that process (see, for example, sections 8 and 10). Within each buffer, there is a position called the *point*, where most of the action takes place.

Emacs displays one or more *windows* into its buffers, each showing some portion of the text of some buffer. A buffer's text is retained even when no window displays it; it can be displayed at any time by giving it a window. Each window has its own point (as just described); when only one window displays a buffer, its point is the same as the buffer's point. Two windows can simultaneously display text (not necessarily the same text) from the same buffer with a different point in each window, although it is most often useful to use multiple windows to display multiple files. At the bottom of each window, Emacs displays a *mode line*, which generally identifies the buffer being displayed and (if applicable) the file associated with it. At any given time, the *cursor*, which generally marks the point of text insertion, is in one of the windows (called the *current window*) at that window's point.

1.2 Commands

At the bottom of Emacs' display is a single *echo area*, displaying the contents of the *minibuffer*. This is a one-line buffer in which one types commands. It is, for many purposes, an ordinary Emacs buffer; standard Emacs text-editing commands for moving left or right and for inserting or deleting characters generally work in it. To issue a command by name, one types `M-x` ("meta-x"; this notation is described below) followed by the name of the command and `RET` (the return key); the echo area displays the command as it is typed. It is only necessary to type as much of the command name as suffices to identify it uniquely. For example, to run the command for looking at a UNIX manual entry—for which the full command is `M-x manual-entry`—it suffices to type `M-x man`, followed by a `RET`.

All Emacs commands have names, and you can issue them with `M-x`. You'll invoke most commands, however, by using control characters and escape sequences to which these commands are *bound*. Almost every character typed to Emacs actually executes a command. By default, typing any of the printable characters executes a command that inserts that character at the cursor. Many of the control characters are bound to commonly-used commands (see the quick-reference guide at the end for a summary of particularly important ones). At any time, it is possible to bind an arbitrary key or sequence of keys to an arbitrary command, thus *customizing* Emacs to your own tastes. Hence, all descriptions of key bindings in this document are actually descriptions of standard or default bindings.

1.3 Notations for special keys

In referring to non-graphic keys (control characters and the like), we'll use the following notations.

`ESC` denotes the escape character.

`DEL` denotes the delete character. On HP workstations, as we've set them up for this class, the '`Backspace`' key has the same effect.

`SPC` denotes the space character.

`RET` denotes the result of pressing the 'Return' key. (Confusingly, the result of typing this into a file is not a return character (ASCII code 13), but rather a linefeed character (ASCII code 10). Nevertheless, Emacs distinguishes the two keys.) On the HP workstations, this is the wide key labelled "Enter" in the main section of the keyboard.

`LFD` denotes the result of typing the linefeed key. On the HP workstations, this is the tall key labelled "Enter" in the numeric keypad at the far right of the keyboard.

`TAB` denotes the tab (also `C-i`) key.

`C-`α denotes "control-α"—the result of holding down the `Control` (or `Ctrl`) key while typing α.

`M-`α denotes "meta-α," which one gets either by typing the two-character sequence `ESC` followed by α, or (on our HP workstations when running the X window system) holding down either `Alt` key while typing α.

`C-M-`α denotes the result of typing the two-character sequence `ESC C-`α, or (on HP workstations when running X) holding down both `Control` and `Alt` simultaneously with typing α).

1.4 Command arguments

Certain commands take arguments, and take these arguments from a variety of sources. Any command may be given a numeric argument. To enter the number comprising the digits $d_0 d_1 \cdots d_n$ as a numeric argument (d_0 may also be a minus sign), type either '`M-`$d_0 d_1 \cdots d_n$' or '`C-u`$d_0 d_1 \cdots d_n$' before the command. When using `C-u`, the digits may be omitted, in which case '4' is assumed. The most common use for numeric arguments is as repetition counts. Thus, `M-4 C-n` moves down four lines and `M-72 *` inserts a line of 72 asterisks in the file. Other commands give other interpretations, as described below. In describing commands, we will use the notation *ARG* to refer to the value of the numeric argument, if present.

When commands prompt for arguments, Emacs will often allow provide a *completion* facility. When entering a file name on the echo line, you can usually save time by typing `TAB`, which fills in as much of the file name as possible, or `SPC` which fills in as much as possible up to a punctuation mark in the file name. Here, "as much as possible" means as much as is possible without having to guess which of several possible names you must have meant. A similar facility will attempt to complete the names of functions or buffers that are prompted for in the echo line.

1.5 Modes

The binding of keys to commands depends on the buffer that currently contains the cursor. This allows different buffers to respond to characters in different ways. In this document, we will refer to the set of key bindings in effect within a given buffer as the *(major) mode* of that buffer (the term "*mode*" is actually somewhat ill-defined in Emacs). A set of key bindings that simply modifies a few characteristics is called a `minor mode`.

Emacs will automatically establish a mode for buffers containing certain files depending on the name of their associated file. Thus, buffers start out in 'C' mode for files whose names end in '`.c`' or '`.h`'; 'C++' mode for `.cc` or `.C`; or 'Scheme' mode (see §9) for `.scm`. These modes affect the behavior of the `TAB` key, for example, causing program text to be indented according to the conventions for a particular programming language. The shell buffer runs in Shell mode, which (among many other things) causes the `RET` key to send the last line typed to the shell. Files with unclassifiable names generally start in Fundamental mode.

There is one useful minor mode that's worth knowing about.

`M-x auto-fill-mode` toggles (reverses the setting) of auto-fill mode, which by default is usually off. In auto-fill mode, lines get broken automatically as they are being typed when they get too long. When you are typing comments in C programs, auto-fill mode will automatically start a new comment on the next line when the current line gets near to filling up.

2 Important special-purpose commands

`C-g` quits the current command. Generally useful for cancelling a `M-x`-style command or other multi-character command that you have started entering. When in doubt, use it.

`C-x C-c` exits from Emacs. It prompts (in the echo area) if there are any buffers that have not been properly saved.

`C-x u` undoes the effects of the last editing command. If repeated, it undoes each of the preceding commands in reverse order (there is a limit). This is an extremely important command; be sure to master it. This does not undo other kinds of commands; the cursor may end up at some rather odd places.

`C-l` redraws the screen, and positions the current line to the center of the current window.

3 Basic Editing

The simple commands in this section will enable you to do most of the text entering and editing that you'll ordinarily need. Periodic browsing through the on-line documentation (see section 7.3) will uncover many more.

3.1 Simple text.

To enter text, simply position the cursor to the desired buffer and character position (using the commands to be described) and type the desired text. Carriage return behaves as you would expect. To enter control characters and other special characters as if they were ordinary characters, precede them with a `C-q`.

3.2 Navigation within a buffer.

The following commands move the cursor within a given buffer. Later sections describe how to move around between buffers.

`C-f` moves forward one character (at the end of a line, this goes to the next).

`C-b` moves backward one character.

`M-f` moves forward one "word."

`M-b` moves backward one word.

`C-e` moves to the end of the current line.

`C-a` moves to the beginning of the current line.

`C-M-f` moves forward one Lisp (Scheme) S-expression.

`C-M-b` moves backward one Lisp (Scheme) S-expression.

`M-a` moves backward to next beginning-of-sentence. The precise meaning of "sentence" depends on the mode.

`M-{` moves backward to next beginning-of-paragraph. The precise meaning of "paragraph" depends on the mode.

`M-e` moves to the next end-of-sentence.

`M-}` moves to the next end-of-paragraph.

`C-n` moves down to the next line (at roughly the same horizontal position, if possible).

`C-p` moves up to the previous line.

`C-v` scrolls the text of the current window up roughly one window-full (i.e., exposes text *later* in the buffer). If *ARG* is supplied, it scrolls up *ARG* lines.

`M-v` scrolls the text of the current window down roughly one window-full (i.e., exposes text *earlier* in the buffer). If *ARG* is supplied, it scrolls down *ARG* lines.

`C-M-v` scrolls up the text in another window (if any) roughly one window-full. If *ARG* is supplied, it scrolls up *ARG* lines.

`M-<` moves to the beginning of the current buffer, after setting the mark (see §3.3) to the current point. If *ARG* is supplied, it moves to a point $ARG/10$ of the way through the buffer, instead of the beginning.

`M->` moves to the end of the current buffer. If *ARG* is supplied, it moves to a point $ARG/10$ of the way back from the end of the buffer, instead of the end.

`M-g` goes to the line number given by the argument (prompts for a number in the echo line, if you haven't supplied an argument).

`M-x what-line` displays the number of the current line in the current buffer.

3.3 Regions

In addition to a point (marked by the cursor in the current window), each buffer may contain a *mark*. Everything between the point and mark is called the *current region*. The current region typically delimits text to be manipulated by certain commands.

`C-@` sets the mark at the current point, and pushes the previous mark on a ring of marks. If *ARG* is present, it instead puts the point at the current mark and pops a new mark off this ring.

`C-SPC` is the same as `C-@`.

`C-x C-x` exchanges the point and the mark.

`M-@` sets the mark after the end of the next word.

`M-h` sets the region (point and mark) around the current paragraph.

`C-x h` sets the region (point and mark) around the entire current buffer.

3.4 Deletion

`DEL` deletes the character preceding the cursor. At the beginning of a line, it deletes the preceding end-of-line character, thus joining the current and preceding lines.

`M-DEL` deletes the word preceding the cursor. The deleted word moves to the kill buffer, described later.

`C-d` deletes the character under the cursor (which can be the end-of-line).

`M-d` deletes the word following the cursor.

`C-k` deletes the rest of the line following the cursor. If the cursor is on the end-of-line, delete the end-of-line. The deleted line moves to the kill buffer.

`M-\` deletes all horizontal blank space on either side of the cursor.

`M-SPC` deletes all but one horizontal blank space surrounding the cursor.

`C-x C-o` on non-blank line, deletes all immediately following blank lines; on isolated blank line, deletes the line; on other blank lines, deletes all but one.

`C-w` deletes everything between the point and the mark, moving the deleted text to the kill buffer.

`M-w` copies everything between point and mark to the kill buffer, without actually deleting it.

3.5 Insertion and the kill buffer

Several of the preceding commands mention the *kill buffer*. Text that is deleted is appended to the end of the current kill buffer, and can later be retrieved and inserted ("pasted" or "yanked") elsewhere in the text (even in another buffer different from its original source). Normally, each time a command that does not append to the kill buffer is executed, the current kill buffer is saved in a ring of kill buffers, and the next deletion command starts with an empty kill buffer. Hence, to move a sequence of lines, one can issue a sequence of `C-k` commands, with no intervening commands, move to the desired destination, and yank them back (with `C-y`).

`C-y` inserts the contents of the current kill buffer at the cursor, and moves cursor to end of inserted text. If a numeric value of ARG is supplied, inserts the ARG^{th} most recent kill buffer in the ring.

`C-u C-y` inserts current kill buffer, as for `C-y`, but leaves point unchanged.

`M-y` when issued *immediately* after a `C-y` or `M-y`, deletes the text inserted by the `C-y` or `M-y` and substitutes the text from the next kill buffer in sequence in the kill ring.

`C-M-w` causes the next command, if a kill command, to append to the end of previous kill buffer, rather than starting with a new one. This allows you, for example, to delete lines from several different places and then yank them back into one place.

3.6 Indentation

Indentation generally depends on the mode of the buffer. When a buffer is associated with a '.scm' file, in particular, it is by default in Scheme mode, in which the standard indentation referred to below is appropriate for Scheme source programs.

`TAB` indents as appropriate for the current mode. In text files, this is just an ordinary typewriter-style tab command. In Scheme source files, it indents to the appropriate point for a standard set of indentation conventions.

`LFD` is the same as `RET TAB`. Thus, if in typing in a Scheme program, you end each line with `LFD` instead of `RET`, your program will be indented as you enter it.

`M-;` indents for a comment according to the current mode. In Scheme mode, this inserts `;`.

`M-LFD` when used inside a comment, will close the comment, if necessary, go to a new line, and start a properly-indented comment on that line.

`C-x TAB` indents the current region "rigidly" by ARG spaces to the right (default 4). Negative arguments indent to the left. Tabs are correctly counted as the appropriate number of blanks.

`C-M-\` indents the current region according to the current mode. For an improperly-indented Scheme program, for example, this will correct all the indentation within the region.

3.7 Miscellaneous manipulations

`C-o` inserts a newline after the cursor. This has the same effect as `RET C-b` (return and then back up one character).

`C-t` transposes the character under the cursor with the preceding character. If an end-of-line is under the cursor, transposes the preceding two characters.

`M-t` transposes the next word that begins left of the cursor with the word following.

`C-x C-t` transposes the current and preceding lines.

`M-c` capitalizes the next word (making all characters other than the first lower case).

`M-u` converts the next word to all upper case.

`M-l` converts the next word to all lower case.

3.8 Using the mouse

When you are using Emacs with the X window system, you may use the mouse for simple positioning, text deletion, and text insertion. The three mouse buttons indicate the operation to be performed, and the mouse pointer (the slanting arrow, which we'll usually just call the *pointer*) usually indicates the position at which to perform it. In the following, the mouse buttons are called 'LB', 'MB', and 'RB', for left button, middle button, and right button. We'll use `C-`*B* to indicate the result of holding down "Control" while pushing button *B*.

`LB` places the point and mark at the position (and in the buffer) indicated by the pointer. You may then drag the mouse with LB depressed; this leaves the mark at the point you pressed `LB` and moves the point (and cursor) to the point at which you release `LB`, thus defining a new current region.

`RB` first extends the current region to include all the text between the existing current region (or the point, if there is no current region) and the pointer. Next, it copies the text in the current region into the kill buffer, as for `M-w` above. When clicked twice for the same text, it also deletes the text. Finally, it also copies the text into something called the *window-system cut buffer*. Text in the window-system cut buffer may be "pasted" (inserted) by `MB`, as described below, not only into Emacs buffers, but also into any other X-windows buffer.

`MB` pastes (inserts) text from the window system cut buffer at the point indicated by the mouse, and puts the cursor at the beginning and the mark at the end of the inserted text. This is somewhat like a mouse version of `C-y`. However, since it takes its text from the window system cut buffer (common to all windows on the screen), it allows the insertion of text from or to a window other than the one running Emacs.

`C-LB` Displays a menu of buffers to move to and allows you to select one (a mouse version of `C-x b`, described later).

You may also use the mouse to select from menus that sprout from the menu bar at the top of your Emacs screen. The content of these menus depends on the kind of buffer you are in.

4 Context searches

The search commands provide a convenient way to position the cursor quickly over long distances. One can search either for specific strings or for patterns specified by *regular expressions*. Both kinds of searches are carried out *incrementally*; that is, as you type in the target string or pattern, the cursor's position is continually changed to point to the first point in the buffer (if any) that matches what you have typed so far.

`C-s` searches forward incrementally.

`C-s C-s` is as for `C-s`, but initialize the search string to the one used in the last string search.

`C-M-s` is as for `C-s`, but searches for a regular expression.

`C-M-s C-s` As for `C-M-s`, but initialize the search pattern to the last pattern used.

`C-r` Search backward incrementally.

`C-r C-r` As for `C-r`, but initialize the search string as for `C-s C-s`.

`M-x occur` prompts for a regular expression and lists each line that follows the point and contains a match for the expression in a buffer. If you give an *ARG*, it will list that number of lines of context around each match.

`M-x count-matches` prompts for a regular expression and displays in the echo area the number of lines following the point that contain a match for it.

`M-x grep` prompts for arguments to the UNIX `grep` utility (which searches files for lines matching a given regular expression) and runs it asynchronously, allowing other editing while the search continues. See the command `` C-x ` `` in §10.1 for a description of how to look at each of the lines found in turn.

`M-x kill-grep` stops a `grep` that was started by `M-x grep`.

As you type the search string or pattern, the cursor moves in the appropriate direction to the first matching string, if any (specifically, to the right end of that string for a forward search and to the left end for a reverse search). By default, the case (upper or lower) of characters is ignored as long as the pattern you type contains no upper-case characters; 'a' will each match either 'a' or 'A'. When the pattern contains at least one upper-case character, the search becomes case-sensitive; 'a' will not match 'A', nor will 'A' match 'a'. If matching fails at any point, you will receive a message to that effect in the echo area. While entering a search string or pattern, certain command characters have altered effects, as follows.

`RET` ends the search, leaving the point at the string found, and setting the mark at the original position of the point.

`DEL` undoes the effect of the last character typed (and not previously DELed), moving the search back to wherever it was previously.

`C-g` aborts the search and returns the cursor to where it was at the beginning of the search.

`C-q` quotes the next character. That is, it causes the next character to be added to the search string or pattern as an ordinary character, ignoring any control action it might normally have. Use this, for example to search for a `C-g` character or, in a regular-expression search, to search for a '.'.

`C-s` begins searching forward at the point of the cursor for the next string satisfying the search string or pattern. If used in a reverse search, therefore, this reverses the sense of the search. If used at the point of a failing search, this starts the search over at the beginning of the buffer ("wraps around").

`C-r` is like `C-s`, but searches in the reverse direction, and can reverse the direction of a forward search.

`C-w` adds the next word beginning at the cursor to the end of the search string or pattern. It follows that this has the effect of moving the cursor forward over that word.

`LFD` adds the rest of the line to the end of the current search string or pattern.

Other control characters terminate the search, and then have their ordinary effect.

Ordinary searches (`C-s` and `C-r`) treat all ordinary characters as search characters. For regular-expression searches, several of these characters have special significance. See also the on-line documentation.

`.` matches any character, except end-of-line.

`^` matches the beginning of a line (that is, it matches the empty string, and only at the beginning of a line.)

`$` matches the end of a line.

`[···]` matches any of the characters between the square brackets. A range of characters may be denoted using '`-`', as in `[a-z0-9]`, which denotes any digit or letter. To include '`]`' as one of the characters, put it first. To include `'-'`, use `'---'`. To include '`^`', do *not* make it the first character.

`[^···]` matches any of the characters `not` included in the '···'. Thus, if end-of-line is not one of the characters, this will match it.

`*` when following another regular expression, denotes zero or more occurrences of that regular expression—in other words, an optional occurrence. This character applies to the immediately preceding regular expression; it has "highest precedence." There are special parentheses (see below) for cases where this is not what you want. Hence, the pattern '`.*`' denotes any number of characters, other than end-of-line. The pattern '`[a-z][a-z0-9_]*`' denotes a letter optionally followed by string of letters, digits, and underscores.

`+` is like '`*`', but denotes at least one occurrence. Thus, '`[0-9]+`' denotes an integer literal.

`?` is like '`*`', but denotes zero or one occurrence. Hence, the pattern '`[0-9]+,?`' denotes an integer literal optionally followed by a comma.

`\(···\)` groups the items '···'. Hence, '`\([0-9]+,\)?`' denotes an optional string consisting of an integer literal followed by a comma. The pattern '`\(01\)*` denotes zero or more occurrences of the two-character string '`01`'.

`\b` matches the empty string at the beginning or end of a word. Hence, '`\bring\b`' matches "ring" standing alone, but not "string" or "rings".

`\B` matches the empty string, provided that it is not at the beginning or end of a word.

`\|` matches a string matching either the regular expression to its left or to its right. Use '`\(\)`' to limit what regular expressions it applies to. Thus, '`\bf[a-z]+\|[0-9]+`' matches any integer literal or any word that begins with 'f', while '`\bf\([a-z]+\|[0-9]+\)`' matches any "word" that begins with 'f' and continues with either all letters or with all digits.

\n where *n* is any digit, denotes the string that matched the pattern within the n^{th} set of '`\(\)`' brackets in the current regular expression. Thus, '`\b\([0-9]+\), *\1`' matches any integer literal that is followed by a comma, an optional space, and a repetition of the same literal; it matches "23, 23" and "10,10", but not "23, 24".

5 Replacement

The following commands allow you to do systematic replacement of one string or pattern with another within a given buffer.

`M-%` performs a query-replace operation. It prompts for a search string and a replacement string. Terminate each of the two with a `RET`. The command will then display each instance of the search string found, and prompt for its disposal. The options are described below. If *ARG* is supplied, it will only match things surrounded by word boundaries, so that if the search string is "top", there will be no replacement inside the string "stop" or "topping".

`M-X query-replace-regexp` is the same as `M-%`, but replaces patterns designated by regular expressions, rather than just simple strings. The replacement string may contain instances of '\n', for *n* a digit, which, as described in the section on regular expressions, denotes the string matched by the n^{th} regular expression in '\(\)' braces in the search string. Thus, for example, the search pattern '`(\([a-z_][a-z0-9_]+\))`' with the replacement pattern '`[\1]`' will replace each C identifier surrounded by parentheses by the same identifier surrounded by square brackets.

By default, the replacement will preserve the case of the letters replaced if the search string or pattern has no upper-case letters, and otherwise will use the case supplied in the replacement string.

At each instance of the search string or pattern, you are prompted for an action. Here are some common ones.

`SPC` replaces the indicated occurrence and goes to the next.

`DEL` keeps the indicated occurrence unchanged and go to the next.

`RET` exits with no further replacements.

`,` makes one replacement, but waits for another `SPC` or `DEL` before moving to the next match.

`.` makes one replacement and then exits.

`!` replaces all remaining occurrences without prompting again.

`?` prints a help message.

`C-r` enters a recursive edit level. That is, you are put back in ordinary Emacs at the point of the current occurrence and can edit in the usual manner. Typing `C-M-c` then goes back to the query-replace command.

`y` same as `SPC`.

`n` same as `DEL`.

`q` same as `RET`.

In addition to replacement, there are two often-useful commands for deleting selected lines.

`M-x delete-matching-lines` prompts for a regular expression and deletes (*without* prompting) each line after the point that contains a match for it.

`M-x delete-non-matching-lines` prompts for a regular expression and deletes each line after the point that does not contain a match for it.

6 Files, buffers, and windows

Each buffer has a name. By default, buffers that are associated with particular files have the name of that file (not including the name of the directory containing it), possibly followed by a number in angle brackets to distinguish multiple files (from different directories with the same name.

6.1 Loading into and storing from buffers

`C-x C-f` prompts for a file name and sets the current window to displaying that file in a buffer having the same name. If a buffer displaying that file already exists, this command merely switches the window to that buffer. If the file does not exist, the buffer is initially empty. The buffer is subsequently associated with the file. This process is called *finding* the file.

`C-x 4 C-f` prompts for a file name, goes to the next window on the screen (creating a new one, if there is only one), and then acts like `C-x C-f`.

`C-x C-s` saves the current buffer in its associated file, if the buffer has been modified. If the file being saved exists, then the old version is first renamed to have a tilde (~) appended to its name, if no such file yet exists.

`C-x C-w` prompts for a file name and saves the current buffer into that file. Generally, it is preferable and safer to use `C-x C-f` or `C-x 4 C-f` and then use `C-x C-s`, but sometimes this command is handy.

`C-x i` prompts for a file name and inserts that file at the point. It does not associate the inserted file with the current buffer.

`M-x revert-buffer` throws away the contents of the current buffer and restores the contents of the associated file. It will ask you to confirm these actions before taking them.

6.2 Manipulating buffers and windows

`C-x o` makes another window on the screen (if any) the current window.

`C-x 0` deletes the current window, expanding another window to take its place. The buffer being displayed in the current window is not affected.

`C-x 1` makes the current window the only window on the screen, deleting all others. The buffers being displayed in the deleted windows are not affected.

`C-x 2` splits the current window into two vertically (one on top of the other), both displaying the same buffer.

`C-x 3` splits the current window into two horizontally (beside each other), each displaying the same buffer.

`C-x b` prompts for a buffer name and switches the current window to that buffer. When trying to move to a buffer associated with a file, it is better to use the file finding commands.

`C-x C-b` lists the active buffers in a window.

`C-x k` prompts for a buffer name and deletes that buffer, displaying some other buffer in the current window. You will be warned if the contents of the buffer have been modified and not yet saved.

6.3 Auto-saving and recovery

Buffers that are associated with files are periodically saved ("auto-saved") in files whose names begin and end with '`#`'. After a crash, you can return yourself to the point at which the last auto-save of a given file took place by using the following command in place of `C-x C-f` or `C-x 4 C-f`.

`M-x recover-file` prompts for a file name, F. It then tries to recover the contents of that file from an auto-save file (named `#`F`#`) in the same directory, if such a file exists and is younger than the any file named F in the directory. After completing this command, `C-x C-s` will save the recovered file to F.

7 On-line documentation

7.1 UNIX documentation

Emacs has a simple interface to the standard UNIX '`man`' command, which provides documentation to UNIX commands:

`M-x manual-entry` prompts for a topic (a UNIX command or subprogram name, usually), and displays the man page for it, if any, in a buffer. The buffer is a perfectly ordinary buffer; you may put the cursor in it and move around using ordinary Emacs navigational commands.

7.2 Basic Emacs help

The help command, `C-h`, provides a variety of useful documentation. The character following `C-h` indicates the specific kind of service desired; the descriptions of several of these follow.

`C-h a` prompts for a pattern (regular expression) and displays a buffer containing all commands whose name contains a match to that pattern, together with a short description and the key sequence to which the command is bound, if any.

`C-h b` displays a buffer containing all bindings of commands to keys. The display is in two parts: the *global bindings* that apply by default in any buffer, and the *local bindings* that apply only when one is in the current buffer, and override any global binding in that buffer.

`C-h f` prompts for a function name and then displays its full documentation in a buffer.

`C-h C-h` documents the help command itself.

`C-h i` runs the 'info' documentation reader (see below).

`C-h k` prompts for a command key sequence and describes the function invoked by that sequence.

`C-h m` prints documentation about the mode of the current buffer.

`C-h t` puts you into an Emacs tutorial.

`C-h w` prompts for a function name and tells what key, if any, invokes it.

7.3 The info browser

The key sequence `C-h i` invokes the documentation browsing system, `info`. Actually, this is little more than a buffer with some special bindings to the keys. Aside from the special bindings, the ordinary Emacs commands will work while inside the `info` buffer. At any time, the `info` buffer, whose name is `*info*`, contains a *node*, a section of text documenting something. These nodes are connected to each other in such a way that one can move quickly from one node to another that covers a related topic. Some nodes contain *menus*, indicated by lines that begin

```
* Menu:
```

The lines after this give the names of other nodes, and descriptions of their contents. One such entry reads as follows.

```
* Commands::      Named functions run by key sequences to do editing.
```

The word(s) between the asterisk and the double-colon name another node. The following key commands, defined only when in the buffer `*info*`, allow one to move through the documentation. They are only a few of the ones provided.

`m` prompts for the name of a node from the menu in the current buffer and displays that node. You need only enter enough to identify the desired entry unambiguously; case is ignored.

`f` follows a cross-reference. Cross references are indicated in the text of a node by a phrase of the form "`* Note` *foo*`::`". One follows them by typing 'f' followed by the name (*foo*) of the referenced node, as for the 'm' command.

`l` goes back to the last-visited node.

`u` goes up to the parent of this node. The definition of parent is actually arbitrary, but is usually a node that contains the current one in its menu.

`d` returns to the top (initial) node of the Info system.

`q` suspends the browser and goes back to where you were when you issued `C-h i`.

`.` returns to the beginning of the text of the current node.

`?` furnishes help about the browser commands.

8 The shell

It is possible to run a UNIX shell under Emacs, and this allows any number of useful effects. The command `M-x shell` moves to a buffer named `*shell*` that is running a UNIX shell (creating it if necessary). Anything typed into this buffer is sent to the shell, just it would be outside of Emacs. Any output produced as a result of the commands sent to the shell is placed at the end of the shell buffer. Because the shell is running in an Emacs window, the contents of the shell can be edited and navigated freely, and the entire record of the input and output to the shell is available at all times. A few keys have slightly different-from-usual meanings in the shell buffer.

`RET` sends whatever line the cursor is on to the shell and moves to the end of the shell buffer. Hence, one can repeat a command by placing the cursor anywhere in it and typing `RET`.

`TAB` attempts to complete the immediately preceding file name.

`C-c C-c` is the same as a single `C-c` outside Emacs.

`C-c C-d` is the same as `C-d` (end-of-file) outside Emacs.

`C-c C-z` is the same as `C-z` outside Emacs.

`C-c C-u` kills the current line of input to the shell.

It is sometimes useful to run a single shell command over a region of text in a buffer.

`M-|` prompts for a shell command and executes it, giving the current region as the standard input. If the `M-|` is preceded by `C-u`, the output of the command replaces the region. Otherwise, the output goes to a separate buffer. For example, to sort the lines in the current region, enter the command `C-u M-| sort`.

9 Running Scheme under Emacs

The best way to run Scheme from a workstation is to do so through Emacs. Just as you can create an Emacs buffer for communicating with a UNIX shell (§8), you can also do so to communicate with a Scheme interpreter. Not only can you interact with the interpreter, but you can also feed files or definitions that you are editing to a running interpreter conveniently without having to load them explicitly.

The command `M-x run-scheme` moves to a buffer named `*scheme*` that is running the Scheme interpreter, creating this buffer if necessary. Each line that you type into this buffer gets sent to the interpreter, just as if you had typed it in while running the interpreter outside of Emacs. Any output from the interpreter in response to your input is appended to the `*scheme*` buffer.

The usual way to create and execute a Scheme program is as follows.

- Using Emacs, create a file to contain your program (or load one that you've already started) using `C-x C-f` or `C-x 4 C-f`; let's suppose the file is named `something.scm` (so that within Emacs, it lives in a buffer of the same name). We have configured Emacs so that any file ending in `.scm` gets edited in Scheme mode, which gives a special meaning to the keys `TAB`, `LFD`, and others described below.
- Edit or add to your file as needed. When typing definitions into the Emacs buffer for `something.scm`, using the `TAB` key at the beginning of each line will automatically indent that line properly. Alternatively, you can end each line by typing `LFD` instead of `RET`; in Scheme mode, `LFD` is short for `RET TAB`. If in the process of editing the buffer, you mess up the indentation of a definition, place the cursor at the beginning of the definition (on or before the opening '(') and type `M-C-q`, which will correctly indent the entire definition.
- Make sure you have a Scheme buffer (named `*scheme*`) running under Emacs (`M-x run-scheme`) will create one if you don't).
- In the buffer for `something.scm`, type `C-c M-l` to load your program into the running Scheme interpreter. Emacs will ask you for a file name; just type `RET`, which will use `something.scm`. If you haven't saved your changes to `something.scm`, Emacs will ask if it should do it for you. The effect of `C-c M-l` is to send the command `(load "something.scm")` to the Scheme interpreter and also to put the cursor in the `*scheme*` buffer, ready to enter Scheme expressions. You'll see the usual response to the `load` command in the `*scheme*` buffer.
- Sometimes—especially when you are correcting a file whose contents you've already loaded into Scheme—it is convenient to send just a single revised definition to the Scheme interpreter. To so do, place the cursor at the beginning of the definition (on or before the opening '(') and type `C-c M-e`. This also puts you into the Scheme buffer.

Here is a concise summary of the Scheme-related commands. These commands are also available from the menu bar. With the exception of `M-x` commands, all of these commands are in effect only in buffers that are in Scheme mode (normally, those containing files whose name ends in `.scm`).

`M-x run-scheme` when used for the first time, creates a buffer named `*scheme*` and runs the Scheme interpreter in it, displaying input from you and output from the interpreter. If the buffer already exists, this command simply moves to that buffer.

`C-c C-z` puts the cursor in the `*scheme*` buffer.

`C-c M-e` sends the definition after the cursor to Scheme (that is, it copies it the `*scheme*` buffer and then sends it to the Scheme interpreter that attached to that buffer). The command also places the cursor at the end of the `*scheme*` buffer.

`C-c C-e` is the same as `C-c M-e`, but leaves the cursor where it is.

`C-c M-l` loads an entire file into Scheme Prompts for a file name; the default is the current buffer's file. Puts the cursor at the end of the `*scheme*` buffer.

`C-c C-l` is the same as `C-c M-l`, but leaves the cursor where it is.

`C-c M-r` sends all the text in the current region to Scheme and puts the cursor at the end of the `*scheme*` buffer.

`C-c C-r` is the same as `C-c M-r`, but leaves the cursor where it is.

`M-C-q` indents the definition after the cursor according to the usual rules for indenting Scheme expressions.

`M-C-\` indents all Scheme expressions in the current region.

`TAB` indents the current line of Scheme code as appropriate for the surrounding context.

`LFD` is the same as `RET TAB`.

10 Compiling, debugging, and tags

[This section is not relevant to CS61A.] Emacs provides rather nice ways of compiling programs, correcting any compilation errors, and debugging the results. It is so much more convenient than entering compilation commands directly from a shell that there is no excuse not to use it.

10.1 Compilation

`M-x compile` prompts for a shell command, and then executes that command in a special buffer, named `*compilation*`. The current file at the time the `M-x compile` is issued determines the directory in which the shell command executes. The default command is simply `make -k`. Assuming you follow the convention of putting an appropriate `make` input file named `makefile` or `Makefile` in each source directory, this command will generally do the right thing. While the compilation proceeds, you are free to edit or use the `*shell*` buffer.

`` C-x ` `` finds the next error message in the buffer `*compilation*` (if any), finds the source files and line referred to by the error message, and displays the error message in one window and the source file in another. Thus, after a compilation is complete (actually, even while it proceeds), you can step through the error messages produced, going automatically to the offending points in the source file so that they can be corrected. The buffer `*compilation*` also contains the output from the `M-x grep` command described in §4.

`M-x kill-compiler` cancels a compilation started by `M-x compile`, if any.

10.2 Using GDB under Emacs

[This section is not relevant to CS61A.] The GNU debugger, GDB, is an interactive source-level debugger for C and several other languages. It can be run under Emacs, which provides a few rather nifty additional features. Full on-line documentation of gdb is available using the `C-h i` command in Emacs. The command `M-x gdb` will prompt for an executable file name, and then run GDB on that file, displaying the interaction in a buffer that acts much like a shell buffer described previously. Within that buffer, however, several commands have a slightly different meaning. In addition, whenever GDB displays the current position in the program (for example, after a step, at a breakpoint, or after an interrupt), Emacs will try to display the indicated source file and line in another window, with an arrow ('`=>`') pointing at the corresponding line in the source text (this arrow is not actually in the file being displayed).

The following commands are peculiar to GDB buffers.

`C-c C-n` performs a GDB 'next' command (step to next line in the source program).

`C-c C-s` performs a GDB 'step' command (step to next line in the source program to be executed, stopping at the beginning of any procedure that gets called.)

`C-c C-i` performs a GDB 'stepi' command (step to next machine-language instruction—not usually used unless you are programming in assembly language.

`C-c <` performs a GDB 'up' command (go up to procedure that called current one).

`C-c >` performs a GDB 'down' command (opposite of 'up').

`C-c C-r` performs a GDB 'finish' command (continues from last breakpoint).

`C-c C-b` set a breakpoint at the current position in the program (as indicated by the position of the '`=>`' arrow).

`C-c C-d` delete a breakpoint (if any) at the current position in the program (as indicated by the position of the '`=>`' arrow).

In addition, within any source file buffer, there is the following command.

`C-x SPC` puts a break point at the point in the program indicated by the cursor.

10.3 Tags

In UNIX terminology, a *tag table* is an index that tells how to find the definition of any certain identifiers ('tags') defined in some collection of source files. In effect, it provides a smart, multi-file search that is particularly useful when navigating in non-trivial directories of source files. Typically, you set things up by going into the directory containing the source text to be indexed and issuing the UNIX command

`etags` *options files*

where *files* is a list of all the source files that need to be indexed. This creates a file named 'TAGS' containing the tag table. For C programs, the tags are the names of functions defined in the named source files. The `-t` option causes `etags` to record **typedef** declarations as well. The tag table produced is organized in such a way that simple edits to a source file will not invalidate it. The following Emacs commands deal with tag tables.

`M-x visit-tags-table` prompts for the name of a tags table file, and uses its contents in future tag-related searches.

`M-.` prompts for a tag and then positions the current window in the file containing its first definition and puts the cursor on that definition. You may also give a null response (just `RET`), in which case the word before or around the point is used as the tag.

`C-u M-.` finds the next alternate definition of the last tag specified.

`C-x 4 .` is the same as `M-.`, but displays the text containing the tag in the other window instead of the current one.

`M-x tags-search` prompts and searches for a regular expression as for `C-M-s`, but is does a non-incremental search through all the files given in the currently-visited tag table.

`M-x tags-query-replace` acts like `M-Q`, but looks through all the files given in the currently-visited tag table.

`M-,` restarts the last `tags-search` or `tags-query-replace` from the current location of the point.

`M-x tags-apropos` prompts for a regular expression and displays a list of all tags in the currently-visited table that match it.

11 But wait; there's more!

As indicated at the beginning, this is not a complete reference manual. It has not covered scrolling sideways, tab setting, the mail system, the Emacs internal Lisp dialect, automatic abbreviation, the spelling checker, the directory editor, the change-log editor, or how to replace all groups of lines of your program that are indented more than *ARG* spaces by '...'[1]. You can learn about these and other topics by using `C-h i`. You might also try typing `C-h f SPC C-x o`, which creates a buffer containing the names of all Emacs functions and then puts the cursor there so that you can scroll through and look for likely-sounding names.

Just use it. Every session is an adventure.

[1]You probably think I'm kidding, don't you? Guess again.