CS61B Tools Documentation
Spring 2006

Paul N. Hilfinger
University of California, Berkeley
Copyright © 2001, 2002, 2004, 2006 by Paul N. Hilfinger. All rights reserved.
# Contents

1. **Highlights of GNU Emacs**  
   1.1 Basic Concepts  
   1.2 Important special-purpose commands  
   1.3 Basic Editing  
   1.3.1 Simple text  
   1.3.2 Navigation within a buffer  
   1.3.3 Context searches  
   1.3.4 Deletion, insertion, and text movement  
   1.3.5 Using the mouse  
   1.3.6 Replacement  
   1.3.7 Modes  
   1.4 Files, buffers, and windows  
   1.5 On-line documentation  
   1.5.1 The info browser  
   1.6 The shell  
   1.7 Compiling, debugging, and tags  
   1.7.1 Compilation  
   1.7.2 Using GDB and GJDB under Emacs  
   1.7.3 Tags  
   1.8 But wait; there's more!  

2. **Basic Compilation: javac and gmake**  
   2.1 Compilation and Interpretation  
   2.2 Where ‘java’ and ‘javac’ find classes  
   2.2.1 The interpreter’s classes  
   2.2.2 The compiler’s classes  
   2.3 Multiple classes in one source file  
   2.4 Compiling multiple files  
   2.5 Archive files  
   2.6 The make utility  
   2.6.1 Basic Operation and Syntax  
   2.6.2 Variables  
   2.6.3 Phony targets  
   2.6.4 Details of actions
2.6.5 Including makefiles .......................... 34

3 The GJDB Debugger .................................. 37
  3.1 Basic functions of a debugger ...................... 37
  3.2 Preparation ..................................... 38
  3.3 Starting GJDB ................................... 38
  3.4 Threads and Frames ................................ 39
  3.5 GJDB Commands .................................. 40
  3.6 Common Problems ................................ 44
  3.7 GJDB use in Emacs ............................... 45
Chapter 1

Highlights of GNU Emacs

This document describes the major features of GNU Emacs (called “Emacs” hereafter), a customizable, self-documenting text editor. There are versions available for all our UNIX systems, as well as Windows 95 and Windows NT. 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 CS61B alternative text editor this semester. I have spoken.

Emacs carries with it on-line documentation of most of its commands, along with a tutorial for first-time users. Section 1.5 describes how to use these facilities. Because this documentation is available, I have not made an attempt to present a complete Emacs reference manual here.

To run Emacs, simply enter the command `emacs` to the shell. If you are on a Sun, it is best to be running under X, so as to get full advantage of the window system. Within Emacs, as described below, you can edit any number of files simultaneously. On UNIX and NT systems, you can also run UNIX shells, and compile, execute, and debug programs. As a result, it should seldom be necessary to leave Emacs before you are ready to logout.

1.1 Basic Concepts

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 1.6 and 1.7). 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.

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 \texttt{M-x} ("meta-x"; this notation is described below) followed by the name of the command and \texttt{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 \texttt{M-x manual-entry}—it suffices to type \texttt{M-x man}, followed by a \texttt{RET}.

All Emacs commands have names, and you can issue them with \texttt{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.

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

\texttt{ESC} denotes the escape character.

\texttt{DEL} denotes the delete character. On HP workstations, the ‘Backspace’ key has the same effect.

\texttt{SPC} denotes the space character.

\texttt{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.)

\texttt{LFD} denotes the result of typing the linefeed key.

\texttt{TAB} denotes the tab (also \texttt{C-i}) key.

\texttt{C-x} denotes the result of control-shifting a character \texttt{x}.

\texttt{M-α} denotes the result of meta-shifting a character α (on our HP workstations when running the X window system, either ‘Alt’ key serves as a meta-shift key; it is held down while typing \texttt{x}). Alternatively, one may type \texttt{M-α} as the two-character sequence \texttt{ESC} followed by \texttt{α}. 

1.2. IMPORTANT SPECIAL-PURPOSE COMMANDS

C-M-α denotes the result of simultaneously control- and meta-shifting x (on HP workstations when running X, hold down the Alt and Control keys simultaneously with typing α). Alternatively, one may type ESC C-α.

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). There are certain standard modes that are described in section 1.3.7.

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 \(d_0d_1\cdots d_n\) as a numeric argument (\(d_0\) may also be a minus sign), type either ‘M-\(d_0d_1\cdots d_n\)’ or ‘C-u\(d_0d_1\cdots d_n\)’ 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.

1.2 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.
1.3 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 1.5.1) will uncover many more.

1.3.1 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.

1.3.2 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).

M-f moves forward one “word.”

C-b moves backward one character.

M-b moves backward one word.

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

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.

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

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 down 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 ‘Regions’ below) 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.

**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. We have set up Emacs so that the current region is shaded.

\( C-@ \) 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.

### 1.3.3 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 section 1.7.1 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”).
1.3. BASIC EDITING

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 nth 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”.

1.3.4 Deletion, insertion, and text movement

The following commands cover most of what you need for local, small edits.

Deleting text.

\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-w is the same as \M-DEL in our version of Emacs. This is not standard, but is provided to avoid confusion with its common use in the shell.

\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.

\M-W deletes everything between the point and the mark. In standard Emacs, this is \C-w, but in our version, this is modified to prevent confusion with the same character in the shell.
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\textsuperscript{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.

M-w is the same as M-w, above, but simply adds the text to the kill buffer without actually deleting it.

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.

Indentation. Indentation generally depends on the mode of the buffer. When a buffer is associated with a file whose *extension* (part after the final period in the file name) is ‘c’ or ‘h’, in particular, it is by default in C mode, in which the standard indentation referred to below is appropriate for C source programs. With an extension of ‘cc’ or ‘C’, it is in C++ mode, and with an extension of ‘java’, it is in Java mode.

TAB indents as appropriate for the current mode. In text files, this is just an ordinary typewriter-style tab command. In C 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 Java 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 C 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.
CHAPTER 1. HIGHLIGHTS OF GNU EMACS

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 C program, for example, this will correct all the indentation within the region.

Other simple 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.

1.3.5 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 puts 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.
1.3. **BASIC EDITING**

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.

1.3.6 **Replacement**

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

\textbf{M-q}  
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”. In standard Emacs, this is M-%.

\textbf{M-Q}  
Is the same as M-q, 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 uppercase 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.

\textbf{SPC}  
Replaces the indicated occurrence and goes to the next.

\textbf{DEL}  
Keeps the indicated occurrence unchanged and go to the next.

\textbf{RET}  
Exits with no further replacements.
CHAPTER 1. HIGHLIGHTS OF GNU EMACS

, makes one replacement, but waits for another \texttt{SPC} or \texttt{DEL} before moving to the next match.

. makes one replacement and then exits.

! replaces all remaining occurrences without prompting again.

? prints a help message.

\texttt{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 \texttt{C-M-c} then goes back to the query-replace command.

\texttt{y} same as \texttt{SPC}.

\texttt{n} same as \texttt{DEL}.

\texttt{q} same as \texttt{RET}.

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

\texttt{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.

\texttt{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.

1.3.7 Modes

Certain collections bindings of keys to commands and other parameter settings are referred to as \textit{modes}. When such collections simply modify a few characteristics, they are called \textit{minor modes}. Emacs will automatically establish a mode for buffers containing certain files depending on the name of their associated file. Thus, buffers with extensions \texttt{.c} and \texttt{.h} start out in C mode, which affects the behavior of tab commands, for example. The shell buffer runs in Shell mode. Files with unclassifiable names generally start in Fundamental mode. For routine work, you will seldom need to worry about these modes.

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

\texttt{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.
1.4 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.

\texttt{C-x C-f} prompts for a file name and sets the current window to displaying that file in a buffer by 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 \textit{finding} the file.

\texttt{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 \texttt{C-x C-f}.

\texttt{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.

\texttt{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 \texttt{C-x C-f} or \texttt{C-x 4 C-f} and then use \texttt{C-x C-s}, but sometimes this command is handy.

\texttt{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.

\texttt{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.

\texttt{C-x o} makes another window on the screen (if any) the current window.

\texttt{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.

\texttt{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.

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

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

\texttt{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.

\texttt{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.

**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.

### 1.5 On-line documentation

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.

In addition, there is a simple interface to the standard UNIX ‘man’ command.
M-x \texttt{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.

\subsection{The info browser}

The key sequence \texttt{C-h i} invokes the documentation browsing system, \texttt{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 \texttt{info} buffer. At any time, the \texttt{info} buffer, whose name is \texttt{*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

\begin{quote}
\texttt{* Menu:}
\end{quote}

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

\begin{quote}
\texttt{* Commands:: Named functions run by key sequences to do editing.}
\end{quote}

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

\begin{itemize}
\item \texttt{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.
\item \texttt{f} follows a cross-reference. Cross references are indicated in the text of a node by a phrase of the form \texttt{"* Note foo::"}. One follows them by typing \texttt{‘f’} followed by the name (\texttt{foo}) of the referenced node, as for the \texttt{‘m’} command.
\item \texttt{l} goes back to the last-visited node.
\item \texttt{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.
\item \texttt{d} returns to the top (initial) node of the Info system.
\item \texttt{q} suspends the browser and goes back to where you were when you issued \texttt{C-h i}.
\item . returns to the beginning of the text of the current node.
\item \texttt{?} furnishes help about the browser commands.
\end{itemize}
1.6 The shell

It is possible to run a UNIX shell under Emacs, and this allows any number of useful effects. The command \texttt{M-x shell} moves to a buffer named \texttt{*shell*} 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.

\texttt{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 \texttt{RET}.

\texttt{TAB} attempts to complete the immediately preceding file name.

\texttt{C-c C-c} is the same as a single \texttt{C-c} outside Emacs.

\texttt{C-c C-d} is the same as \texttt{C-d} (end-of-file) outside Emacs.

\texttt{C-c C-z} is the same as \texttt{C-z} outside Emacs.

\texttt{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.

\texttt{M-|} prompts for a shell command and executes it, giving the current region as the standard input. If the \texttt{M-|} is preceded by \texttt{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 \texttt{C-u M-| sort}.

1.7 Compiling, debugging, and tags

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.

1.7.1 Compilation

\texttt{M-x compile} prompts for a shell command, and then executes that command in a special buffer, named \texttt{*compilation*}. The current file at the time the \texttt{M-x compile} is issued determines the directory in which the shell command executes. The default command is simply \texttt{make -k}. Assuming you follow the convention of putting an appropriate \texttt{make} input file named \texttt{makefile} or \texttt{Makefile} in each source directory, this command will generally “do the right thing” for the files in that directory. While the compilation proceeds, you are free to edit or use the \texttt{*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 section 1.3.3.

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

1.7.2 Using GDB and GJDB under Emacs

The GNU debugger, GDB, is an interactive source-level debugger for C, 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). At any given time, GDB has a notion of the “current call frame” being examined. Initially, this is the function containing the current position in the program, but the ‘up’ and ‘down’ commands will change it “up and down the call chain” to the function that called the current call frame or was called from it. As this happens, the gdb more of Emacs will display the source code around the current call frame.

GJDB is my adaptation of Sun’s rather pitiful Java debugger, jdb, which is distributed with their Java Developer's kit. My version makes it a bit more useful for our purposes, and also sufficiently similar to GDB that this section can apply to both commands. The command M-x gjdb starts GJDB, prompting for the name of the main class—the one containing your main procedure.

The following commands are peculiar to GDB and GJDB buffers.

C-c C-n performs a ‘next’ command, which steps to the next line in the current function.

C-c C-s performs a ‘step’ command, which steps to the next line in the source program to be executed, stopping at the beginning of any function that gets called.

C-c C-i performs a GDB ‘stepi’ command, which steps to the next machine-language instruction. This only makes sense in GDB, not GJDB, and is not usually used unless you are programming in assembly language.

C-c < performs an ‘up’ command, which causes GDB or GJDB to show the caller up to the frame of current frame’s caller.
C-c > performs a ‘down’ command (opposite of ‘up’).

C-c C-r performs a ‘finish’ command (continues from last breakpoint).

C-c C-b sets 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.

(Actually, the official command is C-x C-a C-b, but I can never remember that.)

1.7.3 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 does a non-incremental search through all the files given in the currently-visited tag table.
1.8  BUT WAIT; THERE’S MORE!

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.

1.8  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 ‘...’. 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.

\footnote{You probably think I’m kidding, don’t you? Guess again.}
Chapter 2

Basic Compilation: javac and gmake

[The discussion in this section applies to Java 1.5 tools from Sun Microsystems. Tools from other manufacturers and earlier tools from Sun differ in various details.]

Programming languages do not exist in a vacuum; any actual programming done in any language one does within a programming environment that comprises the various programs, libraries, editors, debuggers, and other tools needed to actually convert program text into action. This document discusses the tools that translate programs into executable form and then execute them.

2.1 Compilation and Interpretation

The Scheme environment that you used in CS61A was particularly simple. It provided a component called the reader, which read in Scheme-program text from files or command lines and converted it into internal Scheme data structures. Then a component called the interpreter operated on these translated programs or statements, performing the actions they denoted. You probably weren’t much aware of the reader; it doesn’t amount to much because of Scheme’s very simple syntax.

Java’s more complex syntax and its static type structure (as discussed in lecture) require that you be a bit more aware of the reader—or compiler, as it is called in the context of Java and most other “production” programming languages. The Java compiler supplied by Sun Microsystems is a program called javac on our systems. You first prepare programs in files (called source files) using any appropriate text editor (Emacs, for example), giving them names that end in ‘.java’. Next you compile them with the java compiler to create new, translated files, called class files, one for each class, with names ending in ‘.class’. Once programs are translated into class files, there is a variety of tools for actually executing them, including Sun’s java interpreter (called ‘java’ on our systems), and interpreters built into products such as Netscape or Internet Explorer. The same class file format works (or is supposed to) on all of these.

In the simplest case, if the class containing your main program or applet is called
CHAPTER 2. BASIC COMPILATION: JAVAC AND GMAKE

C, then you should store it in a file called C.java, and you can compile it with the command

javac C.java

This will produce .class files for C and for any other classes that had to be compiled because they were mentioned (directly or indirectly) in class C. For homework problems, this is often all you need to know, and you can stop reading. However, things rapidly get complicated when a program consists of multiple classes, especially when they occur in multiple packages. In this document, we’ll try to deal with the more straightforward of these complications.

2.2 Where ‘java’ and ‘javac’ find classes

Every Java class resides in a package (a collection of classes and subpackages). For example, the standard class String is actually java.lang.String: the class named String that resides in the subpackage named lang that resides in the outer-level package named java. You use a package declaration at the beginning of a .java source file to indicate what package it is supposed to be in. In the absence of such a declaration, the classes produced from the source file go into the anonymous package, which you can think of as holding all the outer-level packages (such as java).

2.2.1 The interpreter’s classes

When the java program (the interpreter) runs the main procedure in a class, and that main procedure uses some other classes, let’s say A and p.B, the interpreter looks for files A.class and B.class in places that are dictated by things called class paths. Essentially, a class path is a list of directories and archives (see §2.5 below for information on archives). If the interpreter’s class path contains, let’s say, the directories $D_1$ and $D_2$, then upon encountering a mention of class A, java will look for a file named $D_1/A.class$ or $D_2/A.class$. Upon encountering a mention of p.B, it will look for $D_1/p/B.class$ or $D_2/p/B.class$.

The class path is cobbled together from several sources. All Sun’s java tools automatically supply a bootstrap class path, containing the standard libraries and such stuff. If you take no other steps, the only other item on the class path will be the directory ‘.’ (the current directory). Otherwise, if the environment variable CLASSPATH is set, it gets added to the bootstrap class path. In past years, our standard class setup had ‘.’ and the directory containing the ucb package (with our own special classes, lovingly concocted just for you), which we set up with the command

```
setenv CLASSPATH .:/home/ff/cs61b/lib/java/classes
```

(the colon is used in place of comma (for some reason) to separate directory names). The interpreter and compiler would then find the definition of a class such as ucb.io.StdIO in
2.3 Multiple classes in one source file

/home/ff/cs61b/lib/java/classes/ucb/io/StdIO.class

These days, we use archive files instead, as described below in §2.5.

2.2.2 The compiler’s classes

The compiler looks in the same places for .class files, but its life is more complicated, because it also has to find source files. By default, when it needs to find the definition of a class A, it looks for file A.java in the same directories it looks for A.class. This is the easiest case to deal with. If it does not find A.class, it will automatically compile A.java. To use this default behavior, simply make sure that the current directory (‘.’) is in your class path (as it is in our default setup) and put the source for a class A (in the anonymous package) in A.java in the current directory, or for a class p.B in p/B.java, etc., using the commands

```bash
javac A.java
javac p/A.java
```

respectively, to compile them.

It is also possible to put source files, input class files, and output class files (i.e., those created by the compiler) in three different directories, if you really want to (I don’t think we’ll need this). See the -sourcepath and -d options in the on-line documentation for javac, if you are curious.

2.3 Multiple classes in one source file

In general, you should try to put a class named A in a file named A.java (in the appropriate directory). For one thing, this makes it possible for the compiler to find the class’s definition. On the other hand, although public classes must go into files named in this way, other classes don’t really need to. If you have a non-public class that really is used only by class A, then you can put it, too, into A.java. The compiler will still generate a separate .class file for it.

2.4 Compiling multiple files

Java source files depend on each other; that is, the text of one file will refer to definitions in other files. As I said earlier, if you put these source files in the right places, the compiler often will automatically compile all that are needed even if it is only actually asked to compile one “root” class (the one containing the main program or main applet). However, it is possible for the compiler to get confused when (a) some .java files have already been compiled into .class files, and then (b) subsequently changed. Sometimes the compiler will recompile all the necessary files (that is, the ones whose source files have changed or that use classes whose source files have changed), but it is a bit dangerous to rely on this for the Sun compiler. The compiler also can’t find class definitions if you “hide” them by putting, say, several classes into one file. The compiler guesses that class A.B is in file A/B.java.
If it isn’t, then it gives up. You can avoid both of these problems by asking listing all the necessary files for `javac` explicitly:

```bash
javac A.java p/B.java root.java
```

Since this is tedious to write, it is best to rely on a makefile to do it for you, as described below in §2.6.

## 2.5 Archive files

For the purposes of this course, it will be sufficient to have separate `.class` files in appropriate directories, as I have been describing. However in real life, when one’s application consists of large numbers of `.class` files scattered throughout a bunch of directories, it becomes awkward to ship it elsewhere (say to someone attempting to run your Web applet remotely). Therefore, it is also possible to bundle together a bunch of `.class` files into a single file called a *Java archive* (or *jar file*). You can put the name of a jar file as one member of a class path (instead of a directory), and all its member classes will be available just as if they were unpacked into the directory structure described in previous sections.

The utility program `jar`, provided by Sun, can create or examine jar files. Typical usage: to form a jar file `stuff.jar` out of all the classes in package `myPackage`, plus the files `A.class` and `B.class`, use the command

```bash
jar cvf stuff.jar A.class B.class myPackage
```

This assumes that `myPackage` is a subdirectory containing just `.class` files in package `myPackage`. To use this bundle of classes, you might set your class path like this:

```bash
setenv CLASSPATH .:stuff.jar:other directories and archives
```

## 2.6 The make utility

Even relatively small software systems can require rather involved, or at least tedious, sequences of instructions to translate them from source to executable forms. Furthermore, since translation takes time (more than it should) and systems generally come in separately translatable parts, it is desirable to save time by updating only those portions whose source has changed since the last compilation. However, keeping track of and using such information is itself a tedious and error-prone task, if done by hand. Therefore, most programming environments provide some kind of project or compilation-control facility. The UNIX `make` utility is a conceptually simple and general example. It accepts as input a description of the interdependencies of a set of source files and the commands necessary to compile them, known as a `makefile`; it examines the ages of the appropriate files; and it executes whatever commands are necessary, according to the description. For further convenience, it will supply certain standard actions and dependencies by default, making it unnecessary to state them explicitly.
2.6. THE MAKE UTILITY

There are numerous dialects of make, both among UNIX installations and (under other names) in programming environments for personal computers. In this course, I will use a version known as gmake. Though conceptually simple, the gmake utility has accreted features with age and use, and is rather imposing in the glory of its full definition. This document describes only the simple use of gmake.

In addition to compilation (or re-compilation) control, there are other uses for gmake. It is useful in cases where one needs some kind of preprocessing: where the Java source files themselves result from applying some program to other inputs. The makefiles themselves also serve as a useful repository for scripts that perform numerous tasks incidental to compilation. I use them to build course material and copy it to where others can get to it.

2.6.1 Basic Operation and Syntax

Figure 2.1 is a sample makefile for compiling a simple editor program, edit, from eight .java files.

This file consists five rules. A rule consists of a line containing two lists of names separated by a colon, followed by one or more lines beginning with tab characters. Any line may be continued, as illustrated, by putting a backslash at the very end, which essentially acts like a space, combining the line with its successor. The ‘#’ character indicates the start of a comment that goes to the end of that line.

The names preceding the colons are known as targets; they are most often the names of files that are to be produced. The names following the colons are known as dependencies of the targets. They usually denote other files (possibly, other targets) that must be present and up-to-date before the target can be processed. The lines starting with tabs that follow the first line of a rule are called actions. They are shell commands (that is, commands that you could type in response to the Unix prompt) that get executed in order to create or bring up to date the target of the rule (we’ll use the generic term update for the process of determining whether action is necessary on a particular target and then (if needed) building or rebuilding it).

Each rule says, in effect, that to update the targets, each of the dependencies must first be updated (recursively). Next, if a target does not exist (that is, if no file by that name exists) or if it does exist but is older than one of its dependencies (so that one of its dependencies was changed after the target was last updated), the actions of the rule are executed to create or update that target. The program will complain if any dependency does not exist and there is no rule for creating it. To start the process off, the user who executes the gmake utility specifies one or more targets to be updated. The first target of the first rule in the file is the default.

---

1For “GNU make,” GNU being an acronym for “GNU’s Not Unix,” gmake is “copylefted” (it has a license that requires free use of any product containing it). It is also more powerful than the standard make utility.

2Tabs, not blanks. Yes, I know: this is a really irritating design, because if you ever make the mistake of substituting blanks for the tab, you get errors (with very unhelpful messages). The make utility is of rather ancient lineage, and the file format has been this way since before most of you were born (literally).
# Makefile for a simple editor

# The jar file contains the entire collection of classes
# constituting the editor.
edit.jar: edit.class commands.class display.class files.class
    jar cf edit.jar edit.class commands.class display.class files.class

edit.class: edit.java commands.java display.java files.java
    javac -g edit.java commands.java display.java files.java

commands.class: edit.java commands.java display.java files.java
    javac -g edit.java commands.java display.java files.java

display.class: edit.java commands.java display.java files.java
    javac -g edit.java commands.java display.java files.java

files.class: edit.java commands.java display.java files.java
    javac -g edit.java commands.java display.java files.java

Figure 2.1: Sample makefile for an editor program. Adapted from “GNU Make: A Program for Directing Recompilation” by Richard Stallman and Roland McGrath, 1988.

In the example above, edit.jar is the default target. The first step in updating it is to update all the files listed as dependencies (a bunch of .class files). The remaining rules tell how to update each of these .class files. As you can see, they all look pretty much the same, and say that to update X.class:

First update all the source files edit.java, command.java, etc. Next, if X.class is missing or is older than any of these source files, then execute javac on all the source files.

We chose to compile all the source files together like this because otherwise it is possible for the compiler to get confused by old .class files that are still lying around.

Updating the source (.java) files is easy. There are no rules for any of them, so gmake simply insists that they all exist in order to be considered up to date.

Now edit.class, for example, is up to date if it is younger (was created more recently) than all the files edit.java, command.java, and so forth. If instead it is older, gmake assumes that that one of those source files has been changed since the
2.6. THE MAKE UTILITY

last compilation that produced edit.class and must be “rebuilt.” Of course, if edit.class does not exist, then gmake also knows it has to be rebuilt. If rebuilding is necessary, gmake executes the action “javac -g edit.java commands.java ...

producing new .class files. In our particular case, if any one of the .class files needs to be rebuilt, they are all rebuilt. If two of them need to be rebuilt (let’s say edit.class and files.class), then gmake will execute the action for one of them and then, when it checks the other .class file, will discover that it has already been updated. Thus, you needn’t worry that the javac command will be executed more than once.

Once all the .class files are up-to-date, gmake will check to see if any of them are younger than edit.jar (or if edit.jar does not exist). If any of the class files had to be rebuilt, then of course it will be younger, and gmake will execute the indicated action: “jar cf edit.jar....”

To invoke gmake for this example, one issues the command

```
gmake -f makefile-name target-names
```

where the target-names are the targets that you wish to update and the makefile-name given in the -f switch is the name of the makefile. By default, the target is that of the first rule in the file. Furthermore, you may (and usually do) leave off -f makefile-name, in which case it defaults to either Makefile, makefile, or (in the case of gmake only) GNUmakefile, whichever exists. It is typical to arrange that each directory contains the source code for a single principal program. By adopting the convention that the rule with that program as its target goes first, and that the makefile for the directory is named Makefile, you can arrange that, by convention, issuing the command gmake with no arguments in any directory will update the principal program of that directory.

It is possible to have more than one rule with the same target, as long as no more than one rule for each target has an action. Thus, I can also write the latter part of the example above as follows:

```
edit.class:
    javac -g edit.java commands.java display.java files.java

commands.class:
    javac -g edit.java commands.java display.java files.java

display.class:
    javac -g edit.java commands.java display.java files.java

files.class:
    javac -g
```

```
edit.class: edit.java commands.java display.java files.java
commands.class: edit.java commands.java display.java files.java
display.class: edit.java commands.java display.java files.java
files.class: edit.java commands.java display.java files.java
```
The order in which these rules are written is irrelevant. Which order or grouping you choose is largely a matter of taste, aside from which is the first (default) target.

Next, you can combine rules with the same dependencies and action. For example:

```
edit.class commands.class display.class files.class
  javac -g edit.java commands.java display.java files.java
```

or just

```
edit.class commands.class display.class files.class: edit.java \
  commands.java display.java files.java
  javac -g edit.java commands.java display.java files.java
```

The example of this section illustrates the concepts underlying `gmake`. The rest of `gmake`'s features exist mostly to enhance the convenience of using it.

### 2.6.2 Variables

You can clarify the example from §2.6.1 considerably and eliminate redundancy by defining `variables` to contain the names of the files.

```
# Makefile for simple editor

JFLAGS = -g

JAVA_SRCS = edit.java \
  commands.java \
  display.java \
  files.java

CLASSES = edit.class commands.class display.class files.class

edit.jar : $(CLASSES)
  jar cf edit.jar $(CLASSES)

$(CLASSES): $(JAVA_SRCS)
  javac $(JFLAGS) $(JAVA_SRCS)
```

The (continued) line beginning “`JAVA_SRCS` =” defines the variable `JAVA_SRCS`, which can later be referenced as “`$(JAVA_SRCS)`”. These later references cause the definition of `JAVA_SRCS` to be substituted verbatim before the rule is processed. It is somewhat unfortunate that both `gmake` and the shell use ‘$’ to prefix variable references; `gmake` defines ‘$$’ to be simply ‘$’, thus allowing you to send ‘$’s to the shell in actions, where needed.
2.6. THE MAKE UTILITY

You will sometimes find that you need a value that is just like that of some variable, with a certain systematic substitution. For example, given a variable listing the names of all source files, you might want to get the names of all resulting .class files. You can rewrite the definition of CLASSES above to get this.

\[
\text{CLASSES} = $(\text{JAVA_SRCS:.java=.class})
\]

The substitution suffix `:.java=.class` specifies the desired substitution. I now have variables for both the names of all sources and the names of all class files without having to repeat a lot of file names (and possibly make a mistake). (I have assumed here that each source file contains a single class whose name is derived from the source file. You can’t use this trick if that isn’t so.)

Variables may also be set in the command line that invokes \texttt{gmake}. For example, the makefile above contains what might look like an unnecessary definition of JFLAGS. However, defining it like that allows one to write:

\[
\text{gmake JFLAGS=-g -deprecation} \ldots
\]

which passes an extra flag to \texttt{javac} (this one happens to give a fuller explanation of certain warning messages). Variable definitions in the command lines override those in the makefile, which allows the makefile to supply defaults.

2.6.3 Phony targets

It is often useful to have targets for which there are never any corresponding files. If the actions for a target do not create a file by that name, it follows from the definition of how \texttt{gmake} works that the actions for that target will be executed each time \texttt{gmake} is applied to that target (because it will think the target is missing). A common use is to put a standard “clean-up” operation into each of your makefiles, specifying how to get rid of files that can be reconstructed, if necessary. For example, you will often see a rule like this in a makefile.

\[
\text{.PHONY: clean}
\]

\[
\text{clean:}
\]

\[
\text{rm -f *.class *}
\]

Every time you issue the shell command “\texttt{gmake clean},” this action will execute, removing all .class files and Emacs old-version files.

The special .PHONY target tells \texttt{gmake} that \texttt{clean} is not a file, and is instead just the name of a target that is \textit{always} out of date. Therefore, when you make the “clean” target, \texttt{gmake} will always execute the \texttt{rm} command, regardless of what files happen to be lying around. In effect, .PHONY tells \texttt{gmake} to treat \texttt{clean} as a command.

Another possible use is to provide a standard way to run a set of tests on your program—what are typically known as \textit{regression tests}—to see that it is working and has not “regressed” as a result of some change you’ve made. For example, to cause the command
make check

to feed a test file through our editor program and check that it produces the right
result, use:

.PHONY: check

check: edit
    rm -f test-file1
    java edit < test-commands1
    diff test-file1 expected-test-file1

where the test input file test-commands1 presumably contains editor commands
that are supposed to produce a file test-file1, and the file expected-test-file1
contains what is supposed to be in test-file1 after executing those commands.
The first action line of the rule clears away any old copy of test-file1; the second
runs the editor and feeds in test-commands1 through the standard input, and the
third compares the resulting file with its expected contents. If either the second or
third action fails, gmake will report that it encountered an error.

2.6.4 Details of actions

By default, each action line specified in a rule is executed by the Bourne shell (as
opposed to the C shell, which, most unfortunately, is more commonly used here).
For the simple makefiles we are likely to use, this will make little difference, but be
prepared for surprises if you get ambitious.

The gmake program usually prints each action as it is executed, but there are
times when this is not desirable. Therefore, a '@' character at the beginning of an
action suppresses the default printing. Here is an example of a common use.

edit.jar : $(CLASSES)
    @echo Creating edit.jar ... 
    @jar cf edit.jar $(CLASSES)
    @echo Done

The result of these actions is that when gmake executes this final step for the edit
program, the only thing you’ll see printed is a line reading “Creating edit.jar
...” and, at the end of the step, a line reading “Done”.

When gmake encounters an action that returns a non-zero exit code, the UNIX
convention for indicating an error, its standard response is to end processing and
exit. The error codes of action lines that begin with a ‘-’ sign (possibly preceded by
a ‘@’ ) are ignored. Also, the -k switch to gmake will cause it to abandon processing
only of the current rule (and any that depend on its target) upon encountering an
error, allowing processing of “sibling” rules to proceed.

2.6.5 Including makefiles

A good way to create makefiles is to have a template that you include in your
particular makefile—something like the example in Figure 2.2. We’ve prepared one
like this already, so that in the very simplest case, your makefile can contain just:
2.6. THE MAKE UTILITY

JAVA_SRCS = edit.java commands.java display.java files.java

include $(MASTERDIR)/lib/java.Makefile.std

As you can probably guess, the include line is a special command that essentially gets replaced by the contents of the named file.

Figure 2.2 illustrates what such a template file might look like. It uses one obscure new feature that makes it possible to partially define an action, and allow others to add to it. The definition of the phony target clean uses two colons rather than one. This is a signal that there may be other “double-colon” rules for clean, complete with actions. They will all get used (in the order encountered). For example, if you include this particular template in a place where you want to define additional clean-up actions besides the ones defined in the template, you can write:

JAVA_SRCS = edit.java commands.java display.java files.java

include $(MASTERDIR)/lib/java.Makefile.std

clean::
   rm -rf test-output

which will cause gmake clean to remove the directory test-output as well as the class files and Emacs-generated files removed in the template.
# Standard definitions for make utility: Java version.

# Assumes that this file is included from a Makefile that defines
# JAVA_SRCS to be a list of Java source files to be compiled.
# It may optionally define OTHER_CLASSES to contain names of classes
# that aren't derivable from the names of the JAVA_SRCS files.
# The including Makefile may subsequently override JFLAGS (flags to
# the Java compiler), and JAVAC (the Java compiler's name), by putting
# these definitions after the "include".

# Targets defined:
# default: Default entry. Compiles classes from all source files.
# clean:: Remove back-up files and files that make can reconstruct.
# You can add additional clean-up actions by adding more
# 'clean::' targets (note the double colon) to your makefile.
# check: Look in the subdirectory tests for all files whose name ends
# in ' .sh'. Each of these should be an executable shell script
# (a file of commands such as you could enter at the command
# prompt) that performs some test of the program. Run each
# and report all that fail (return a non-zero exit code).

JAVAC = javac
JFLAGS = -g
CLASSES = $(JAVA_SRCS:.java=.class) $(OTHER_CLASSES)

.PHONY: clean check default

# Default entry
default: $(CLASSES)

$(CLASSES): $(JAVA_SRCS)

$(JAVAC) $(JFLAGS) $(JAVA_SRCS)
clean::
/bin/rm -f $(CLASSES) *-

check:: $(CLASSES)

cd tests; for test in *.sh; do 
  if ./$$test; then \n    echo "$$test: OK."; \n  else \n    echo "$$test: FAILED."; \n  fi; 
done

Figure 2.2: An example of a file of standard makefile definitions that can be included
from a specific makefile to compile many simple collections of Java programs.
Chapter 3

The GJDB Debugger

A debugger is a program that runs other programs, allowing its user to exercise some degree of control over these programs, and to examine them when things go amiss. Sun Microsystems, Inc. distributes a text-based debugger, called JDB, with its Java Developer’s Kit (JDK). I have modified JDB to make its commands look pretty much like GDB, the GNU Debugger\(^1\), which handles C, C++, Pascal, Ada, and a number of other languages. The result is called GJDB (g’jay dee bee). Perhaps the most convenient way to use it is through the interface supplied with Emacs.

GJDB is dauntingly chock-full of useful stuff, but for our purposes, a small set of its features will suffice. This document describes them.

3.1 Basic functions of a debugger

When you are executing a program containing errors that manifest themselves during execution, there are several things you might want to do or know.

- What statement or expression was the program executing at the time of a fatal error?

- If a fatal error occurs while executing a function, what line of the program contains the call to that function?

- What are the values of program variables (including parameters) at a particular point during execution of the program?

- What is the result of evaluating a particular expression at some point in the program?

- What is the sequence of statements actually executed in a program?

- When does the value of a particular variable change?

\(^1\)The recursive acronym GNU means “GNU’s Not Unix” and refers to a larger project to provide free software tools.
CHAPTER 3. THE GJDB DEBUGGER

These functions require that the user of a debugger be able to examine program data, to obtain a traceback (a list of function calls that are currently executing sorted by who called whom), to set breakpoints where execution of the program is suspended to allow its data to be examined, and to step through the statements of a program to see what actually happens. GJDB provides all these functions. It is a symbolic or source-level debugger, creating the fiction that you are executing the Java statements in your source program rather than the machine code they have actually been translated into.

3.2 Preparation

In this course, we use a system that compiles (translates) Java programs into executable files containing bytecode, a sort of machine language for an idealized virtual machine that is considerably easier to execute than the original source text. This translation process generally loses information about the original Java statements that were translated. A single Java statement usually translates to several machine statements, and most local variable names are simply eliminated. Information about actual variable names and about the original Java statements in your source program is unnecessary for simply executing your program. Therefore, for a source-level debugger to work properly, the compiler must retain some of this superfluous information (superfluous, that is, for execution).

To indicate to our compiler (javac) that you intend to debug your program, and therefore need this extra information, add the -g switch during both compilation.

For example, if you are compiling an application whose main class is called Main, you might compile with

```
javac -g Main.java
```

This sample command sequence produces a class file `Main.class` containing the translation of the class `Main`, and possibly some other class files.

3.3 Starting GJDB

To run this under control of gjdb, you can type

```
gjdb Main
```

in a shell. You will be rewarded with the initial command prompt:

```
[-]
```

This provides an effective, but unfrilly text interface to the debugger. I don’t actually recommend that you do this; it’s much better to use the Emacs facilities described below. However, the text interface will do for describing the commands.
3.4 Threads and Frames

When GJDB starts, your program has not started; it won’t until you tell GJDB to run it (you tell the program is not started from GJDB’s prompt, which will be [-]). After the program has started and before it exits, GJDB will see a set of threads, each one of which is essentially a semi-independent program. If you haven’t encountered Java threads before, the part of your program that you usually think of as “the program” will be the main thread, appropriately named main. However, there will also be a bunch of system threads (running various support activities), that GJDB will tell you about if asked, but which will generally not be of interest. GJDB can examine one thread at a time; which one being indicated by the prompt:

[-] Means there are no threads; the program has not been started.

[?] Means the program is started, but GJDB is not looking at any particular thread. You’ll often see this if you interrupt your program.

name[n] Means that GJDB is looking at thread name, and at frame #n (see below) within that thread.

At any given time, a particular thread is in the process of executing some statement inside a function (method). To arrive inside that method, the program had to execute a method call in a statement of some other method (or possibly the same, in the case of recursion), and so on back to the mysterious system magic that started it all. In other words, in each thread, there is a sequence of currently active method calls, each of which is executing a particular statement, and each of which also has a bunch of other associated information: parameter values, local variable values and so forth. We refer to each of these active calls as frames, or sometimes stack frames, because they come and go in last-in-first-out order, like a stack data structure. Each has a current location, which is a statement or piece of a statement that is currently being executed in that call (sometimes called a program counter or, confusingly, PC). The most recent, or top frame is the one that is executing “the next statement in the program,” while each of the other frames is executing a (so-far incomplete) method call.

For example, consider the simple class Example on page 40. Suppose we start the program with command-line argument 5, and are stopped at statement (E). Then (for the main thread) GJDB sees frames #0–#5, as follows:

<table>
<thead>
<tr>
<th>Frame#</th>
<th>Method</th>
<th>Location</th>
<th>Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.</td>
<td>report</td>
<td>(E)</td>
<td>x: 2</td>
</tr>
<tr>
<td>1.</td>
<td>ilog</td>
<td>(C)</td>
<td>x: 1, a: 2</td>
</tr>
<tr>
<td>2.</td>
<td>ilog</td>
<td>(D)</td>
<td>x: 2, a: 1</td>
</tr>
<tr>
<td>3.</td>
<td>ilog</td>
<td>(D)</td>
<td>x: 5, a: 0</td>
</tr>
<tr>
<td>4.</td>
<td>process</td>
<td>(B)</td>
<td>x: &quot;5&quot;</td>
</tr>
<tr>
<td>5.</td>
<td>main</td>
<td>(A)</td>
<td>args: { &quot;5&quot; }</td>
</tr>
</tbody>
</table>

\[2\]Even when your program is initializing a field in a record, which doesn’t look as if it’s inside a method, it is actually executing a part of either a constructor or a special “static initializer” method (which you’ll see in certain listings under the name <clinit>).
class Example {
    public static void main (String[] args) {
        for (int i = 0; i < args.length; i += 1)
            process (args[i]); // (A)
    }

    void process (String x) {
        ilog (Integer.parseInt(x), 0); // (B)
    }

    void ilog (int x, int a) {
        if (x <= 1)
            report (a); // (C)
        else
            ilog (x/2, a+1); // (D)
    }

    int report (int x) {
        System.out.println (x); // (E)
    }
}

3.5 GJDB Commands

Whenever the command prompt appears, you have available the following commands. Actually, you can abbreviate most of them with a sufficiently long prefix. For example, p is short for print, and b is short for break.

help command
   Provide a brief description of a GJDB command or topic. Plain help lists the possible topics.

run command-line-arguments
   Starts your program as if you had typed

        java Main command-line-arguments

   to a Unix shell. GJDB remembers the arguments you pass, and plain run thereafter will restart your program from the top with those arguments. By default, the standard input to your program will come from the terminal (which causes some conflict with entering debugging commands: see below). However, you may take the standard input from an arbitrary file by using input redirection: adding < filename to the end of the command-line-arguments uses the contents of the named file as the standard input (as it does for the shell). Likewise, adding > filename causes the standard output from your program to go to the named file rather than to the terminal, and >& filename causes
3.5. **GJDB COMMANDS**

both the standard output and the standard error output to go to the named file.

**where**
Produce a backtrace—the chain of function calls that brought the program to its current place. The commands `bt` and `backtrace` are synonyms.

**up**
Move the current frame that GJDB is examining to the caller of that frame. Very often, your program will blow up in a library function—one for which there is no source code available, such as one of the I/O routines. You will need to do several `ups` to get to the last point in your program that was actually executing. Emacs (see below) provides the shorthand `C-c<` (Control-C followed by less-than), or the function key `f3`.

**up n** Perform `n` `up` commands (\(n\) a positive number).

**down**
Undoes the effect of one `up`. Emacs provides the shorthands `C-c>` and function key `f4`.

**down n** Perform `n` `down` commands (\(n\) a positive number).

**frame n** Perform `ups` or `downs` as needed to make frame \#\(n\) the current frame.

**thread n** Make thread \#\(n\) (as reported by `info threads`, below) the current thread that GJDB is examining.

**print \(E\)**
prints the value of \(E\) in the current frame in the program, where \(E\) is a Java expression (often just a variable). For example

```
main[0] print A[i]
A[i] = -14
main[0] print A[i]+x
A[i]+Main.x = 17
```

This tells us that the value of \(A[i]\) in the current frame is -14 and that when this value is added to `Main.x`, it gives 17. Printing a reference value is less informative:

```
main[0] p args
args = instance of java.lang.String[3] (id=172)
```

This tells you that `args` contains a pointer to a 3-element array of strings, but not what these strings are.

**print/\(n\) \(E\)** also prints the value of expression \(E\) in the current frame. If \(E\) is a reference value, however, it also prints the subcomponents (fields or array
elements) of the referenced object to \( n \) levels. Plain \texttt{print} without this specification is equivalent to \texttt{print/0}, and does not print subcomponents. \textit{Printing subcomponents to one level} means printing each subcomponent of \( E \)'s value as if by \texttt{print/0}. Printing to two levels prints means printing each subcomponent as if by \texttt{print/1}, and so forth recursively. For example,

\begin{verbatim}
main[0] print/1 args
args = instance of java.lang.String[3] (id=172) {
  "A", "B", "C"
}
main[0] p T
T = instance of Tree(id=176)
main[0] p/1 T
T = instance of Tree(id=176) {
  label: "A"
  left: null
  right: instance of Tree(id=178)
}
main[0] p/2 T
T = instance of Tree(id=176) {
  label: "A"
  left: null
  right: instance of Tree(id=178) {
    label: "B"
    left: null
    right: instance of Tree(id=180)
  }
}

dump \( E \)
  Equivalent to \texttt{print/1} \( E \).

dump/n \( E \)
  Equivalent to \texttt{print/n} \( E \).

\texttt{info locals} Print the values of all parameters and local variables in the current frame.

\texttt{info threads} List all current threads.

\texttt{quit}
  Leave GJDB.

The commands to this point give you enough to pinpoint where your program blows up, and usually to find the offending bad pointer or array index that is the immediate cause of the problem (of course, the actual error probably occurred much earlier in the program; that’s why debugging is not completely automatic.) Personally, I usually don’t need more than this; once I know where my program goes wrong, I
often have enough clues to narrow down my search for the error. You should at least establish the place of a catastrophic error before seeking someone else’s assistance.

The next bunch of commands allow you to actively stop a program during normal operation.

**suspend and C-f**

When a program is run from a Unix shell, C-c will terminate its execution (usually). At the moment, unfortunately, it will also do this to GJDB itself. When debugging, you usually want instead to simply stop the debugged program temporarily in order to examine it. When the standard input is redirected from a file (using `<'; see the run command), you can simply use suspend to stop the program (and then use continue or resume to restart). When the program is running and standard input comes from the terminal, things get complicated: how does GJDB know a command from program input. If you are using GJDB mode (see §3.7), then C-c C-c will do the trick in this case. Otherwise, if you are running in an ordinary shell, use C-f following by return. And finally, if you are running in a shell under Emacs, use C-qC-f followed by return.

**break place**

Establishes a breakpoint; the program will halt when it gets there. The easiest breakpoints to set are at the beginnings of functions, as in

```
[-] break Example.process
Set breakpoint request Example:8
```

(using the class Example from §3.4). Use the full method name (complete with class and package qualification), as shown. You will either get a confirming message as above (saying that the system set a breakpoint at line 8 of the file containing class Example), or something like

```
Deferring BP RatioCalc.main [unresolved].
It will be set after the class is loaded.
```

when you set a breakpoint before the class in question has been loaded. Breakpoints in anonymous classes are a bit tricky; their names generally have the form “C$n” where C is the name of the outermost class enclosing them, and n is some integer. The problem is that you don’t generally know the value of n. GJDB therefore allows “C.0” as a class name, meaning “any anonymous class inside C.”

When you run your program and it hits a breakpoint, you’ll get a message and prompt like this.

```
Breakpoint hit: thread="main", Example.main(), line=4, bci=22
main[0]
```

(Here, “bci” indicates a position within the bytecode translation of the method; it is not generally very useful). Emacs allows you to set breakpoints with the mouse (see §3.7).
condition $N \ cond$ Make breakpoint number $N$ conditional, so that the program only stops if $cond$, which must be a boolean expression, evaluates to true.

c-condition $N$ Make breakpoint number $N$ unconditional.

delete
Removes breakpoints. This form of the command gives you a choice of breakpoints to delete, and is generally most convenient.

c-cont or continue
Continues regular execution of the program from a breakpoint or other stop.

step
Executes the current line of the program and stops on the next statement to be executed.

next
Like step, however if the current line of the program contains a function call (so that step would stop at the beginning of that function), does not stop in that function.

finish
Does nexts, without stopping, until the current method (frame) exits.

3.6 Common Problems

Name unknown. When you see responses like this:

```java
main[0] print x
Name unknown: x
main[0] print f(3)
Name unknown: f
```

check to see if the variable or method in question is static. A current limitation of the debugger is that you must fully qualify such names with the class that defines them, as in

```
main[0] print Example.f(3)
```

Beware also that fully qualified names include the package name.

Ignoring breakpoints. For a variety of reasons, it is possible for a program to miss a breakpoint that you thought you had set. Unfortunately, GJDB is not terribly good at the moment at catching certain errors. In particular, it will tell you that a breakpoint has been deferred, when in fact it will never be hit due to a class name being misspelled.
3.7 GJDB use in Emacs

While one *can* use `gjdb` from a shell, nobody in his right mind would want to do so. *Emacs* provides a much better interface that saves an enormous amount of typing, mouse-moving, and general confusion. Executing the Emacs command `M-x gjdb` starts up a new window running `gjdb`, and enables a number of Emacs shortcuts, as well as providing a *Debug* menu for issuing many GJDB commands. This command prompts for a command string (typically `gjdb classname`) and (for certain historical reasons) creates a buffer named `*gud-classname*`. Emacs intercepts output from `gjdb` and interprets it for you. When you stop at a breakpoint, Emacs will take the file and line number reported by `gjdb`, and display the file contents, with the point of the breakpoint (or error) marked. As you step through a program, likewise, Emacs will follow your progress in the source file. Other commands allow you to set or delete breakpoints at positions indicated by the mouse.

The following table describes the available commands. On the left, you’ll find the text command line, as described in §3.5. Next comes the *Debug* menu button (if any) that invokes the command. This menu applies both to the GJDB buffer and to buffers containing `.java` files. Next come the Emacs shortcuts: sequences of keys that run the commands. The shortcuts are slightly different in the GJDB buffer as opposed to buffers containing source (`.java`) files, so there are two columns of shortcuts. The last column contains further description. Finally, here are a few reminders about Emacs terminology:

1. In shortcuts, `C-x` means “control-x,” `S-x` means “shift-x,” `fn` refers to one of the function keys (typically above the keyboard), and `SPC` is the space character.

2. The *point*, in Emacs, refers to the location of the cursor; there is one for each buffer. You can set the point using the usual motion commands when in the buffer, or by simply clicking the mouse at the desired spot.

3. The *region* in any given buffer is a section of text (usually shadowed or highlighted so that you can tell where it is). One convenient way to set it is by dragging the mouse over the text you want included while holding down the left mouse button.
### Table 3.1: Summary of Commands for Program Control

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Emacs</th>
<th>GJDB buffer</th>
<th>.java buffer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
<td>Step Over</td>
<td>f6 or C-c</td>
<td>f6</td>
<td>Execute to the next statement of the program; if this statement contains function calls, execute them completely before stopping. [See Note 3, below]</td>
</tr>
<tr>
<td>step</td>
<td>Step Into</td>
<td>f5, or C-c C-s</td>
<td>f5</td>
<td>Execute to the next statement of the program; if this statement calls a function, stop at its first line. [See Note 3, below]</td>
</tr>
<tr>
<td>finish</td>
<td>Finish Function</td>
<td>f7 or C-c C-f</td>
<td>f7</td>
<td>Execute until the current function call returns.</td>
</tr>
<tr>
<td>continue</td>
<td>Continue</td>
<td>f8 or C-c C-r</td>
<td>f8</td>
<td>Continue execution of stopped program.</td>
</tr>
<tr>
<td>suspend</td>
<td>Interrupt</td>
<td>C-c C-c</td>
<td></td>
<td>Interrupt execution of program and suspend its threads.</td>
</tr>
<tr>
<td>C-f</td>
<td>Interrupt</td>
<td>C-c C-c</td>
<td></td>
<td>Same as suspend, but works in cases where the debugged program is running and GJDB is passing input to it from the terminal.</td>
</tr>
<tr>
<td>break file:line#</td>
<td>Set Breakpoint</td>
<td>C-x SPC</td>
<td></td>
<td>Set a breakpoint at the point (applies only to the source buffer).</td>
</tr>
<tr>
<td>delete file:line#</td>
<td>Clear Breakpoint</td>
<td></td>
<td></td>
<td>Remove a breakpoint at the point (applies only to the source buffer).</td>
</tr>
<tr>
<td>run</td>
<td>Run</td>
<td></td>
<td></td>
<td>(Re)start the program, using the last set of command-line arguments. Only available in the GJDB buffer.</td>
</tr>
<tr>
<td>quit</td>
<td>Quit</td>
<td></td>
<td></td>
<td>Leave GJDB. Only available in the GJDB buffer.</td>
</tr>
<tr>
<td>-</td>
<td>Refresh</td>
<td></td>
<td></td>
<td>Re-arrange Emacs’ windows as needed to display the current source line that GJDB is looking at.</td>
</tr>
<tr>
<td>-</td>
<td>Start Debugger</td>
<td></td>
<td></td>
<td>Run gjdb on the class in this (source) buffer.</td>
</tr>
</tbody>
</table>
### 3.7. GJDB USE IN EMACS

<table>
<thead>
<tr>
<th>Command Line</th>
<th>Menu</th>
<th>Emacs</th>
<th>GJDB buffer</th>
<th>.java buffer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>print expr</code></td>
<td>Print</td>
<td>f9</td>
<td>f9</td>
<td></td>
<td>Evaluate <code>expr</code> and print, without showing any subcomponents of the value. Emacs commands apply either to the contents of the region, or if it is inactive, to the variable, field selection, or function call at or after the point.</td>
</tr>
<tr>
<td><code>dump expr</code></td>
<td>Print Details</td>
<td>S-f9</td>
<td>S-f9</td>
<td></td>
<td>Evaluate <code>expr</code> and print, also printing any components (array elements or fields). With Emacs, gets the expression to print as for <code>print</code>.</td>
</tr>
<tr>
<td><code>info locals</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Print (as for the <code>print</code> command) the values of all local variables in the current frame.</td>
</tr>
<tr>
<td><code>up</code></td>
<td>View Caller</td>
<td>f3 or C-c &lt;</td>
<td>f3</td>
<td></td>
<td>Move the debugger’s current focus of attention up one frame; if looking at frame <code>n</code> at the moment, we switch to frame <code>n - 1</code>.</td>
</tr>
<tr>
<td><code>down</code></td>
<td>View Callee</td>
<td>f4 or C-c &gt;</td>
<td>f4</td>
<td></td>
<td>Move the debugger’s current focus of attention down one frame (from frame <code>n</code> to frame <code>n + 1</code>). Opposite of <code>up</code>.</td>
</tr>
<tr>
<td><code>where</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Print a backtrace, showing all active subprogram calls.</td>
</tr>
<tr>
<td><code>info threads</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>List all threads in the program.</td>
</tr>
<tr>
<td><code>thread N</code></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Make thread <code>#N</code> be the one that GJDB is currently examining.</td>
</tr>
</tbody>
</table>