Viper Is a Package for Emacs Rebels

a Vi emulator for GNU Emacs
June 1996, Viper Version 2.90

Masahiko Sato (VIP 3.5)
Aamod Sane (VIP 4.4)
Michael Kifer (Viper)

Distribution

Copyright © 1995, 1996 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the same conditions as for modified versions.

Introduction

We believe that one or more of the following statements are adequate descriptions:

```
Viper Is a Package for Emacs Rebels;
it is a VI Plan for Emacs Rescue
and/or a venomous VI PERil.
```

Technically speaking, Viper is a Vi emulation package for Emacs 19 and XEmacs 19. Because of its reliance on minor mode keymaps, it will not work under Emacs 18. Viper contains virtually all of Vi and Ex functionality and much more. It gives you the best of both worlds: Vi keystrokes for editing combined with the GNU Emacs environment. Viper also fixes some common complaints with Vi commands. This manual describes Viper, concentrating on the differences from Vi and on the new features of Viper.

Viper, formerly known as VIP-19, was written by Michael Kifer. It is based on VIP version 3.5 by Masahiko Sato and VIP version 4.4 by Aamod Sane. Viper tries to be compatible with these packages.

Viper is intended to be usable without reading this manual — the defaults are set to make Viper as close to Vi as possible. At startup, Viper will attempt to set the most appropriate default environment for you, based on your familiarity with Emacs. It will also tell you the basic GNU Emacs window management commands to help you start immediately.

Although this manual explains how to customize Viper, some basic familiarity with Emacs Lisp would be a plus.

It is recommended that you read the chapter Overview. The other chapters will be useful for customization and advanced usage.

You should also learn to use the Info on-line hypertext manual system that comes with Emacs. This manual can be read as an Info file. Try the command *`ESC x info`* with vanilla Emacs sometime.

Comments and bug reports are welcome. `kifer@cs.sunysb.edu` is the current address for Viper bug reports. Please use the Ex command *`:submitReport`* for this purpose.

1 Overview of Viper

Viper is a Vi emulation on top of Emacs. At the same time, Viper provides a virtually unrestricted access to Emacs facilities. Perfect compatibility with Vi is possible but not desirable. This chapter tells you about the Emacs ideas that you should know about, how to use Viper within Emacs and some incompatibilities.

Viper was formerly known as VIP-19, which was a descendant of VIP 3.5 by Masahiko Sato and VIP 4.4 by Aamod Sane.

1.1 Emacs Preliminaries

Emacs can edit several files at once. A file in Emacs is placed in a *buffer* that usually has the same name as the file. Buffers are also used for other purposes, such as shell interfaces, directory editing, etc. See section “Directory Editor” in *The GNU Emacs Manual*, for an example.

A buffer has a distinguished position called the *point*. A *point* is always between 2 characters, and is *looking at* the right hand character. The cursor is positioned on the right hand character. Thus, when the *point* is looking at the end-of-line, the cursor is on the end-of-line character, i.e. beyond the last character on the line. This is the default Emacs behavior.

The default settings of Viper try to mimic the behavior of Vi, preventing the cursor from going beyond the last character on the line. By using Emacs commands directly (such as those bound to arrow keys), it is possible to get the cursor beyond the end-of-line. However, this won’t (or shouldn’t) happen if you restrict yourself to standard Vi keys, unless you modify the default editing style. See Chapter 3 [Customization], page 21.

In addition to the *point*, there is another distinguished buffer position called the *mark*. See section “Mark” in *The GNU Emacs manual*, for more info on the mark. The text between the *point* and the *mark* is called the *region* of the buffer. For the Viper user, this simply means that in addition to the Vi textmarkers a–z, there is another marker called *mark*. This is similar to the unnamed Vi marker used by the jump commands (“ and ”), which move the cursor to the position of the last absolute jump. Viper provides access to the region in most text manipulation commands as `r` and `R` suffix to commands that operate on text regions, e.g., `dr` to delete region, etc. See Section 2.1 [Basics], page 13, for more info.

Emacs divides the screen into tiled *windows*. You can see the contents of a buffer through the window associated with the buffer. The cursor of the screen is positioned on the character after *point*. Every window has a *mode line* that displays information about the buffer. You can change the format of the mode line, but normally if you see ‘`**`’ at the beginning of a mode line it means that the buffer is *modified*. If you write out the contents of a buffer to a file, then the buffer will become not modified. Also if you see ‘`%%`’ at the beginning of the mode line, it means that the file associated with the buffer is write protected. The mode line will also show the buffer name and current major and minor modes (see below). A special buffer called *Minibuffer* is displayed as the last line in a Minibuffer window. The Minibuffer window is used for command input output. Viper uses Minibuffer window for `/` and `:` commands.

An Emacs buffer can have a *major mode* that customizes Emacs for editing text of a particular sort by changing the functionality of the keys. Keys are defined using a *keymap* that records the bindings between keystrokes and functions. The *global keymap* is common to all the buffers. Additionally, each buffer has its *local keymap* that determines the *mode* of the buffer. If a function is bound to some key in the local keymap then that function will be executed when you type the key. If no function is bound to a key in the local map, however, the function bound to the key in the global map will be executed. See section "Major Modes" in *The GNU Emacs Manual*, for more information.

A buffer can also have a *minor mode*. Minor modes are options that you can use or not. A buffer in `text-mode` can have `auto-fill-mode` as minor mode, which can be turned off or on at any time. In Emacs 19, a minor mode may have it own keymap, which overrides the local keymap when the minor mode is turned on. For more information, see section "Minor Modes" in *The GNU Emacs Manual*

Viper is implemented as a collection of minor modes. Different minor modes are involved when Viper emulates Vi command mode, Vi insert mode, etc. You can also turn Viper on and off at any time while in Vi command mode. See Section 1.3 [States in Viper], page 5, for more information.

Emacs uses Control and Meta modifiers. These are denoted as C and M, e.g. `^Z` as `C-z` and `Meta-x` as `M-x`. The Meta key is usually located on each side of the Space bar; it is used in a manner similar to the Control key, e.g., `M-x` means typing `x` while holding the Meta key down. For keyboards that do not have a Meta key, ⟨ESC⟩ is used as Meta. Thus `M-x` is typed as `ESC x`. Viper uses ⟨ESC⟩ to switch from Insert state to Vi state. Therefore Viper defines `_` as its Meta key in Vi state. See Section 1.3.2 [Vi State], page 7, for more info.

Emacs is structured as a lisp interpreter around a C core. Emacs keys cause lisp functions to be called. It is possible to call these functions directly, by typing `M-x function-name`.

1.2 Loading Viper

First, make sure that all six Viper files '`viper*.el`' are somewhere on Emacs *load path*. The load path is a list of directories where Emacs looks for its Lisp code; it is similar to Unix environment variable *PATH*. The load path is determined by the Lisp variable `load-path`.

Type `C-h v load-path RET` to check the current load path of your Emacs. (Here and later, `RET` stands for carriage return.) If the files '`viper*.el`' are not in any directory on the load path, you should create your own directory and put it on the load path; then put '`viper*.el`' in that directory. To put a new directory, say '`~/emacslib`', on your load path, add the following line at the beginning of your '`~/.emacs`' file:

```
(setq load-path (cons "~/emacslib" load-path))
```

It is recommended that you compile '`viper.el`' by running

```
make all
```

in the directory where Viper was unpacked. The best way to ensure that Viper is installed properly is to run

```
make install
```

in the directory where Viper was unpacked. See README in the Viper distribution for the details on which modifications to the make file may be necessary.

Once Viper is settled on the load path, the most common way to load it automatically is to include the line:

```
(require 'viper)
```

in your '`~/.emacs`' file. The '`.emacs`' file is placed in your home directory and it will be executed every time you invoke Emacs. Viper also uses the file '`~/.vip`' for Viper-specific customization. If you wish to be in Vi command state whenever this is appropriate, you can include the following line in your '`.vip`':

```
(setq vip-always t)
```

(See Section 1.3.2 [Vi State], page 7, for the explanation of Vi mode.)

Once invoked, Viper will arrange to bring up Emacs buffers in Vi state whenever this makes sense. See Section 3.2.1 [Packages that Change Keymaps], page 29, to find out when forcing Vi command state on a buffer may be counter-productive.

Even if your '`.emacs`' and '`.vip`' files do not contain any of the above lines, you can still load Viper and enter Vi state by typing the following from within Emacs:

```
M-x load-library RET viper RET
M-x viper-mode
```

Ask your local Emacs administrator if this fails to work.

When Emacs first comes up, if you have not specified a file on the command line, it will show the '`*scratch*`' buffer, in the '`Lisp Interaction`' mode. After you invoke Viper, you can start editing files by using *`:e`*, *`:vi`*, or *`v`* commands. (See Section 4.4 [File and Buffer Handling], page 51, for more information on *`v`* and other new commands that, in many cases, are more convenient than *`:e`*, *`:vi`*, and similar old-style Vi commands.)

1.3 States in Viper

Viper has four states, Emacs, Vi, Insert, and Replace.

'`Emacs state`'

This is the mode plain vanilla Emacs is normally in. After you have loaded Viper, *`C-z`* will normally take you to Vi command state. Another *`C-z`* will take you back to Emacs state. This toggle key can be changed, see Chapter 3 [Customization], page 21 You can also type *`M-x viper-mode`* to change to Vi state.

For users who chose to set their user level to 1 at Viper setup time, switching to Emacs state is deliberately made harder in order to not confuse the novice user. In this case, *`C-z`* will either iconify Emacs (if Emacs runs as an application under X Windows) or it will stop Emacs (if Emacs runs on a dumb terminal or in an Xterm window).

'`Vi state`' This is the Vi command mode. Any of the Vi commands, such as *`i, o, a`*, ..., will take you to Insert state. All Vi commands may be used in this mode.

Most Ex commands can also be used. For a full list of Ex commands supported by Viper, type *:* and then *TAB*. To get help on any issue, including the Ex commands, type *:help*. This will invoke Viper Info (if it is installed). Then typing *i* will prompt you for a topic to search in the index. Note, to search for Ex commands in the index, you should start them with a "*:*", e.g., *:WW*.

'`Insert state`'

Insert state is the Vi insertion mode. *ESC* will take you back to Vi state. Insert state editing can be done, including auto-indentation. By default, Viper disables Emacs keybindings in Insert state.

'`Replace state`'

Commands like *cw* invoke the Replace state. When you cross the boundary of a replacement region (usually designated via a '`$`' sign), it will automatically change to Insert state. You do not have to worry about it. The key bindings remain practically the same as in Insert state. If you type ESC, Viper will switch to Vi command mode, terminating the replacement state.

The modes are indicated on the *mode line* as <E>, <I>, <V>, and <R>, so that the multiple modes do not confuse you. Most of your editing can be done in Vi and Insert states. Viper will try to make all new buffers be in Vi state, but sometimes they may come up in Emacs state. *C-z* will take you to Vi state in such a case. In some major modes, like Dired, Info, Gnus, etc., you should not switch to Vi state (and Viper will not attempt to do so) because these modes are not intended for text editing and many of the Vi keys have special meaning there. If you plan to read news, browse directories, read mail, etc., from Emacs (which you should start doing soon!), you should learn about the meaning of the various keys in those special modes (typing *C-h m* in a buffer provides help with key bindings for the major mode of that buffer).

If you switch to Vi in Dired or similar modes—no harm is done. It is just that the special keybindings provided by those modes will be temporarily overshadowed by Viper's bindings. Switching back to Viper's Emacs state will revive the environment provided by the current major mode.

States in Viper are orthogonal to Emacs major modes, such as C mode or Dired mode. You can turn Viper on and off for any Emacs state. When Viper is turned on, Vi state can be used to move around. In Insert state, the bindings for these modes can be accessed. For beginners (users at Viper levels 1 and 2), these bindings are suppressed in Insert state, so that new users are not confused by the Emacs states. Note that unless you allow Emacs bindings in Insert state, you cannot do many interesting things, like language sensitive editing. For the novice user (at Viper level 1), all major mode bindings are turned off in Vi state as well. This includes the bindings for key sequences that start with *C-c*, which practically means that all major mode bindings are supported. See Chapter 3 [Customization], page 21, to find out how to allow Emacs keys in Insert state.

1.3.1 Emacs State

You will be in this mode only by accident (hopefully). This is the state Emacs is normally in (imagine!!). Now leave it as soon as possible by typing *`C-z`*. The you will be in Vi state (sigh of relief) :-).

Emacs state is actually a Viperism to denote all the major and minor modes (See Section 1.1 [Emacs Preliminaries], page 3) other than Viper that Emacs can be in. Emacs can have several modes, such as C mode for editing C programs, LaTeX mode for editing LaTeX documents, Dired for directory editing, etc. These are major modes, each with a different set of key-bindings. Viper states are orthogonal to these Emacs major modes. The presence of these language sensitive and other modes is a major win over Vi. See Chapter 2 [Improvements over Vi], page 13, for more.

The bindings for these modes can be made available in the Viper Insert state as well as in Emacs state. Unless you specify your user level as 1 (a novice), all major mode key sequences that start with *`C-x`* and *`C-c`* are also available in Vi state. This is important because major modes designed for editing files, such as cc-mode or latex-mode, use key sequences that begin with *`C-x`* and *`C-c`*.

There is also a key that lets you temporarily escape to Vi command state from Emacs state: typing *`M-C-z`* (or *`ESC C-z`*) will let you execute a single Vi command while staying in Viper's Emacs state. At present, this works only for 1-character Vi commands (and for some complex commands).

1.3.2 Vi State

This is the Vi command mode. When Viper is in Vi state, you will see the sign <V> in the mode line. Most keys will work as in Vi. The notable exceptions are:

`C-x`
: *`C-x`* is used to invoke Emacs commands, mainly those that do window management. *`C-x 2`* will split a window, *`C-x 0`* will close a window. *`C-x 1`* will close all other windows. *`C-xb`* is used to switch buffers in a window, and *`C-xo`* to move through windows. These are about the only necessary keystrokes. For the rest, see the GNU Emacs Manual.

`C-c`
: For user levels 2 and higher, this key serves as a prefix key for the key sequences used by various major modes. For users at Viper level 1, *`C-c`* simply beeps.

`C-g and C-]`
: These are the Emacs '`quit`' keys. There will be cases where you will have to use *`C-g`* to quit. Similarly, *`C-]`* is used to exit '`Recursive Edits`' in Emacs for which there is no comparable Vi functionality and no key-binding. Recursive edits are indicated by '`[]`' brackets framing the modes on the mode line. See section "Recursive Edit" in *The GNU Emacs Manual*. At user level 1, *`C-g`* is bound to `vip-info-on-file` function instead.

`Underscore`
: Viper uses *`ESC`* as a switch between Insert and Vi states. Emacs uses *`ESC`* for Meta. We need a Meta key to call the Meta key functions such as *`M-x function name`*. The underscore key, `_`, was chosen, since it is the only reasonable, free

Vi key left. Thus, in Vi state, to get `M-x`, you should type `_x` (if the keyboard has no Meta key). It is possible to use `ESC` as Meta, but then you cannot press `ESC` multiple times in Vi state. See Chapter 3 [Customization], page 21, to find out how to rebind ESC to be Meta.

Other differences are mostly improvements. The ones you should know about are:

‘`Undo`’ `u` will undo. Undo can be repeated by the `.` key. Undo itself can be undone. Another `u` will change the direction. The presence of repeatable undo means that `U`, undoing lines, is not very important. Therefore, `U` also calls `vip-undo`.

‘`Counts`’ Most commands, `~`, `[[`, `p`, `/`, . . ., etc., take counts.

‘`Regexps`’ Viper uses Emacs Regular Expressions for searches. These are a superset of Vi regular expressions, excepting the change-of-case escapes ‘`\u`’, ‘`\L`’, . . ., etc. See section “Regular Expressions” in *The GNU Emacs Manual*, for details. Files specified to `:e` use `csh` regular expressions (globbing, wildcards, what have you). However, the function `vip-toggle-search-style`, bound to `C-c /`, lets the user switch from search with regular expressions to plain vanilla search and vice versa. It also lets one switch from case-sensitive search to case-insensitive and back. See Section 3.3 [Viper Specials], page 31, for more details.

‘`Ex commands`’

The current working directory of a buffer is automatically inserted in the minibuffer if you type `:e` then space. Absolute filenames are required less often in Viper. For path names, Emacs uses a convention that is slightly different from that of Unix. It is designed to minimize the need for deleting path names that Emacs provides in its prompts. (This is usually convenient, but occasionally the prompt may suggest a wrong path name for you.) If you see a prompt `/usr/foo/` and you wish to edit the file `~/.vip`, you don’t have to erase the prompt. Instead, simply continue typing what you need. Emacs will interpret `/usr/foo/~/.vip` correctly. Similarly, if the prompt is `~/foo/` and you need to get to `/bar/file`, keep typing. Emacs interprets `~/foo//bar/` as `/bar/file`, since when it sees ‘`//`’, it understands that `~/foo/` is to be discarded.

The command `:cd` will change the default directory for the current buffer. The command `:e` will interpret the filename argument in `csh`. See Chapter 3 [Customization], page 21, if you want to change the default shell. The command `:next` takes counts from `:args`, so that `:rew` is obsolete. Also, `:args` will show only the invisible files (i.e., those that are not currently seen in Emacs windows).

When applicable, Ex commands support file completion and history. This means that by typing a partial file name and then `TAB`, Emacs will try to complete the name or it will offer a menu of possible completions. This works similarly to Tcsh and extends the behavior of Csh. While Emacs is waiting for a file name, you can type `M-p` to get the previous file name you typed. Repeatedly typing `M-p` and `M-n` will let you browse through the file history.

Like file names, partially typed Ex commands can be completed by typing `TAB`, and Viper keeps the history of Ex commands. After typing `:`, you can browse through the previously entered Ex commands by typing `M-p` and `M-n`. Viper

> tries to rationalize when it puts Ex commands on the history list. For instance, if you typed `:w! foo`, only `:w!` will be placed on the history list. This is because the last history element is the default that can be invoked simply by typing `: RET`. If `:w! foo` were placed on the list, it would be all to easy to override valuable data in another file. Reconstructing the full command, `:w! foo`, from the history is still not that hard, since Viper has a separate history for file names. By typing `: M-p`, you will get `:w!` in the Minibuffer. Then, repeated `M-p` will get you through the file history, inserting one file name after another.
>
> In contrast to `:w! foo`, if the command were `:r foo`, the entire command will appear in the history list. This is because having `:r` alone as a default is meaningless, since this command requires a file argument.

As Vi, Viper's destructive commands can be re-executed by typing '`.`'. However, in addition, Viper keeps track of the history of such commands. This history can be perused by typing `C-c M-p` and `C-c M-n`. Having found the appropriate command, it can be then executed by typing '`.`'. See Chapter 2 [Improvements over Vi], page 13, for more information.

1.3.3 Insert State

To avoid confusing the beginner (at Viper level 1 and 2), Viper makes only the standard Vi keys available in Insert state. The implication is that Emacs major modes cannot be used Insert state. It is strongly recommended that as soon as you are comfortable, make the Emacs state bindings visible (by changing your user level to 3 or higher). See Chapter 3 [Customization], page 21, to see how to do this.

Once this is done, it is possible to do quite a bit of editing in Insert state. For instance, Emacs has a *yank* command, `C-y`, which is similar to Vi's `p`. However, unlike `p`, `C-y` can be used in Insert state of Viper. Emacs also has a kill ring where it keeps pieces of text you deleted while editing buffers. The command `M-y` is used to delete the text previously put back by Emacs' `C-y` or by Vi's `p` command and reinsert text that was placed on the kill-ring earlier.

This works both in Vi and Insert states. In Vi state, `M-y` is a much better alternative to the usual Vi's way of recovering the 10 previously deleted chunks of text. In Insert state, you can use this as follows. Suppose you deleted a piece of text and now you need to re-insert it while editing in Insert mode. The key `C-y` will put back the most recently deleted chunk. If this is not what you want, type `M-y` repeatedly and, hopefully, you will find the chunk you want.

Finally, in Insert and Replace states, Viper provides the history of pieces of text inserted in previous insert or replace commands. These strings of text can be recovered by repeatedly typing `C-c M-p` or `C-c M-n` while in Insert or Replace state. (This feature is disabled in the minibuffer: the above keys are usually bound to other histories, which are more appropriate in the minibuffer.)

You can call Meta functions from Insert state. The Vi-to-Emacs toggle key, normally `C-z`, is bound to Meta in Insert state. Thus `M-x` is typed as `C-zx`. The key `_` will now insert itself—it is not a Meta key in Insert state. (It is well understood that switching Meta keys around is inconvenient. However, there seems to be no way around this problem, short of changing Vi ESC key to something else. See Chapter 3 [Customization], page 21, for the information on `vip-ESC-key`, which can be used to free up *ESC* and designate some other key to do escape from Insert state to Vi state.)

Other Emacs commands that are useful in Insert state are *C-e* and *C-a*, which move the cursor to the end and the beginning of the current line, respectively. You can also use *M-f* and *M-b*, which move the cursor forward (or backward) one word. If your display has a Meta key, these functions are invoked by holding the Meta key and then typing *f* and *b*, respectively. On displays without the Meta key, these functions are invoked by typing *C-z f* and *C-z b* (*C-z* simulates the Meta key in Insert state, as explained above).

When Viper is in Insert state, you will see <I> in the mode line.

1.3.4 Replace State

This state is entered through Vi replacement commands, such as *C*, *cw*, etc., or by typing *R*. In Replace state, Viper puts <R> in the mode line to let you know which state is in effect. If Replace state is entered through *R*, Viper stays in that state until the user hits ESC. If this state is entered via the other replacement commands, then Replace state is in effect until you hit *ESC* or until you cross the rightmost boundary of the replacement region. In the latter case, Viper changes its state from Replace to Insert (which you will notice by the change in the mode line).

Since Viper runs under Emacs, it is possible to switch between buffers while in Replace state. You can also move the cursor using the arrow keys (even on dumb terminals!) and the mouse. Because of this freedom (which is unattainable in regular Vi), it is possible to take the cursor outside the replacement region. (This may be necessary for several reasons, including the need to enable text selection and region-setting with the mouse.)

The issue then arises as to what to do when the user hits the ESC key. In Vi, this would cause the text between cursor and the end of the replacement region to be deleted. But what if, as is possible in Viper, the cursor is not inside the replacement region?

To solve the problem, Viper keeps track of the last cursor position while it was still inside the replacement region. So, in the above situation, Viper would delete text between this position and the end of the replacement region.

1.4 The Minibuffer

The Minibuffer is where commands are entered in. Editing can be done by commands from Insert state, namely:

C-h	Backspace
C-w	Delete Word
C-u	Erase line
C-v	Quote the following character
RET	Execute command

`C-g` and `C-]`

Emacs quit and abort keys. These may be necessary. See Section 1.3.2 [Vi State], page 7, for an explanation.

`M-p` and `M-n`

These keys are bound to functions that peruse minibuffer history. The precise history to be perused depends on the context. It may be the history of search strings, Ex commands, file names, etc.

Most of the Emacs keys are functional in the Minibuffer. While in the Minibuffer, Viper tries to make editing resemble Vi's behavior when the latter is waiting for the user to type an Ex command. In particular, you can use the regular Vi commands to edit the Minibuffer. You can switch between the Vi state and Insert state at will, and even use the replace mode. Initially, the Minibuffer comes up in Insert state.

Some users prefer plain Emacs bindings in the Minibuffer. To this end, set `vip-vi-style-in-minibuffer` to `nil` in '`.vip`'. See Chapter 3 [Customization], page 21, to learn how to do this.

When the Minibuffer changes Viper states, you will notice that the appearance of the text there changes as well. This is useful because the Minibuffer has no mode line to tell which Vi state it is in. The appearance of the text in the Minibuffer can be changed. See Section 3.3 [Viper Specials], page 31, for more details.

1.5 Multiple Files in Viper

Viper can edit multiple files. This means, for example that you never need to suffer through `No write since last change` errors. Some Viper elements are common over all the files.

'`Textmarkers`'

Textmarkers remember *files and positions*. If you set marker '`a`' in file '`foo`', start editing file '`bar`' and type *`'a`*, then *YOU WILL SWITCH TO FILE* '`foo`'. You can see the contents of a textmarker using the Viper command *`[<a-z>`* where <a-z> are the textmarkers, e.g., *`[a`* to view marker '`a`' .

'`Repeated Commands`'

Command repetitions are common over files. Typing *`!!`* will repeat the last *`!`* command whichever file it was issued from. Typing `.` will repeat the last command from any file, and searches will repeat the last search. Ex commands can be repeated by typing *`:RET`*.Note: in some rare cases, that *`:RET`* may do something dangerous. However, usually its effect can be undone by typing *`u`*.

'`Registers`'

Registers are common to files. Also, text yanked with *`y`* can be put back (*`p`*) into any file. The Viper command *`]<a-z>`*, where <a-z> are the registers, can be used to look at the contents of a register, e.g., type *`]a`* to view register '`a`'.

There is one difference in text deletion that you should be aware of. This difference comes from Emacs and was adopted in Viper because we find it very useful. In Vi, if you delete a line, say, and then another line, these two deletions are separated and are put back separately if you use the '`p`' command. In Emacs (and Viper), successive series of deletions that are *not interrupted* by other commands are lumped together, so the deleted text gets accumulated and can be put back as one chunk. If you want to break a sequence of deletions so that the newly deleted text could be put back separately from the previously deleted text, you should perform a non-deleting action, e.g., move the cursor one character in any direction.

'`Absolute Filenames`'

The current directory name for a file is automatically prepended to the file name in any *`:e`*, *`:r`*, *`:w`*, etc., command (in Emacs, each buffer has a current directory). This directory is inserted in the Minibuffer once you type space after *`:e, r`*, etc. Viper also supports completion of file names and Ex commands (*`TAB`*), and it keeps track of command and file history (*`M-p`*, *`M-n`*). Absolute filenames are required less often in Viper.

You should be aware that Emacs interprets *`/foo/bar//bla`* as *`/bla`* and *`/foo/~/bar`* as *`~/bar`*. This is designed to minimize the need for erasing path names that Emacs suggests in its prompts, if a suggested path name is not what you wanted.

The command *`:cd`* will change the default directory for the current Emacs buffer. The Ex command *`:e`* will interpret the filename argument in '`csh`', by default. See Chapter 3 [Customization], page 21, if you want to change this.

Currently undisplayed files can be listed using the *`:ar`* command. The command *`:n`* can be given counts from the *`:ar`* list to switch to other files.

1.6 Unimplemented Features

Unimplemented features include:

- *`:ab`* and *`:una`* are not implemented. Both *`:map`* and *`:ab`* are considered obsolete, since Emacs has much more powerful facilities for defining keyboard macros and abbreviations.
- *`:set option?`* is not implemented. The current *`:set`* can also be used to set Emacs variables.
- *`:se list`* requires modification of the display code for Emacs, so it is not implemented. A useful alternative is `cat -t -e file`. Unfortunately, it cannot be used directly inside Emacs, since Emacs will obdurately change '`^I`' back to normal tabs.

2 Improvements over Vi

Some common problems with Vi and Ex have been solved in Viper. This includes better implementation of existing commands, new commands, and the facilities provided by Emacs.

2.1 Basics

The Vi command set is based on the idea of combining motion commands with other commands. The motion command is used as a text region specifier for other commands. We classify motion commands into *point commands* and *line commands*.

The point commands are:

> `h`, `l`, `0`, `$`, `w`, `W`, `b`, `B`, `e`, `E`, `(`, `)`, `/`, `?`, `‘`, `f`, `F`, `t`, `T`, `%`, `;`, `,`, `^`

The line commands are:

> `j`, `k`, `+`, `-`, `H`, `M`, `L`, `{`, `}`, `G`, `’`, `[[`, `]]`, `[]`

If a point command is given as an argument to a modifying command, the region determined by the point command will be affected by the modifying command. On the other hand, if a line command is given as an argument to a modifying command, the region determined by the line command will be enlarged so that it will become the smallest region properly containing the region and consisting of whole lines (we call this process *expanding the region*), and then the enlarged region will be affected by the modifying command. Text Deletion Commands (See Section 4.2.5 [Deleting Text], page 47), Change commands (See Section 4.2.6 [Changing Text], page 47), even Shell Commands (See Section 4.6 [Shell Commands], page 53) use these commands to describe a region of text to operate on. Thus, type `dw` to delete a word, `>}` to shift a paragraph, or `!’afmt` to format a region from ‘`point`’ to textmarker ‘`a`’.

Viper adds the region specifiers ‘`r`’ and ‘`R`’. Emacs has a special marker called *mark*. The text-area between the current cursor position *point* and the *mark* is called the *region*. ‘`r`’ specifies the raw region and ‘`R`’ is the expanded region (i.e., the minimal contiguous chunk of full lines that contains the raw region). `dr` will now delete the region, `>r` will shift it, etc. `r,R` are not motion commands, however. The special mark is set by `m.` and other commands. See Section 4.2.2 [Marking], page 45, for more info.

Viper also adds counts to most commands for which it would make sense.

In the Overview chapter, some Multiple File issues were discussed (See Section 1.5 [Multiple Files in Viper], page 11). In addition to the files, Emacs has buffers. These can be seen in the `:args` list and switched using `:next` if you type `:set ex-cycle-through-non-files t`, or specify `(setq ex-cycle-through-non-files t)` in your ‘`.vip`’ file. See Chapter 3 [Customization], page 21, for details.

2.2 Undo and Backups

Viper provides multiple undo. The number of undo's and the size is limited by the machine. The Viper command *u* does an undo. Undo can be repeated by typing *.* (a period). Another *u* will undo the undo, and further *.* will repeat it. Typing *u* does the first undo, and changes the direction.

Since the undo size is limited, Viper can create backup files and auto-save files. It will normally do this automatically. It is possible to have numbered backups, etc. For details, see section "Backup and Auto-Save" in *The GNU Emacs Manual*

The results of the 9 previous changes are available in the 9 numeric registers, as in Vi. The extra goody is the ability to *view* these registers, in addition to being able to access them through *p* and *M-y* (See Section 1.3.3 [Insert State], page 9, for details.) The Viper command *] register* will display the contents of any register, numeric or alphabetical. The related command *[textmarker* will show the text around the textmarker. '`register`' and '`textmarker`' can be any letters from a through z.

2.3 History

History is provided for Ex commands, Vi searches, file names, pieces of text inserted in earlier commands that use Insert or Replace state, and for destructive commands in Vi state. These are useful for fixing those small typos that screw up searches and *:s*, and for eliminating routine associated with repeated typing of file names or pieces of text that need to be inserted frequently. At the *:* or */* prompts in the Minibuffer, you can do the following:

M-p and M-n
: To move to previous and next history items. This causes the history items to appear on the command line, where you can edit them, or simply type Return to execute.

M-r and M-s
: To search backward and forward through the history.

RET
: Type RET to accept a default (which is displayed in the prompt).

The history of insertions can be perused by typing *C-c M-p* and *C-c M-n* while in Insert or Replace state. The history of destructive Vi commands can be perused via the same keys when Viper is in Vi state. See Section 3.3 [Viper Specials], page 31, for details.

All Ex commands have a file history. For instance, typing *:e*, space and then *M-p* will bring up the name of the previously typed file name. Repeatedly typing *M-p*, *M-n*, etc., will let you browse through the file history.

Similarly, commands that have to do with switching buffers have a buffer history, and commands that expect strings or regular expressions keep a history on those items.

2.4 Macros and Registers

Viper facilitates the use of Emacs-style keyboard macros. `@#` will start a macro definition. As you type, the commands will be executed, and remembered (This is called “learn mode” in some editors.) `@register` will complete the macro, putting it into ‘`register`’, where ‘`register`’ is any character from ‘`a`’ through ‘`z`’. Then you can execute this macro using `@register`. It is, of course, possible to yank some text into a register and execute it using `@register`. Typing `@@`, `@RET`, or `@LF` will execute the last macro that was executed using `@register`.

Viper will automatically lowercase the register, so that pressing the `SHIFT` key for `@` will not create problems. This is for `@` macros and `"p` *only*. In the case of *y*, `"Ayy` will append to *register a*. For `` [,],',` ``, it is an error to use a Uppercase register name.

The contents of a register can be seen by `]register`. (`[textmarker` will show the contents of a textmarker).

The last keyboard macro can also be executed using `*`, and it can be yanked into a register using `@!register`. This is useful for Emacs style keyboard macros defined using `C-x(` and `C-x)`. Emacs keyboard macros have more capabilities. See section “Keyboard Macros” in *The GNU Emacs Manual*, for details.

Keyboard Macros allow an interesting form of Query-Replace: `/pattern` or `n` to go to the next pattern (the query), followed by a Keyboard Macro execution `@@` (the replace).

Viper also provides Vi-style macros. See Section 3.4 [Vi Macros], page 36, for details.

2.5 Completion

Completion is done when you type `TAB`. The Emacs completer does not grok wildcards in filenames. Once you type a wildcard, the completer will no longer work for that path. Remember that Emacs interprets a file name of the form `/foo//bar` as `/bar` and `/foo/~/bar` as `~/bar`.

2.6 Improved Search

Viper provides buffer search, the ability to search the buffer for a region under the cursor. You have to turn this on in ‘`.vip`’ either by calling

```
(vip-buffer-search-enable)
```

or by setting `vip-buffer-search-char` to, say, `f3`:

```
(setq vip-buffer-search-char [f3])
```

If the user calls `vip-buffer-search-enable` explicitly (the first method), then `vip-buffer-search-char` will be set to `g`. Regardless of how this feature is enabled, the key `vip-buffer-search-char` will take movement commands, like `w,/,e`, to find a region and then search for the contents of that region. This command is very useful for searching for variable names, etc., in a program. The search can be repeated by `n` or reversed by `N`.

Emacs provides incremental search. As you type the string in, the cursor will move to the next match. You can snarf words from the buffer as you go along. Incremental Search is normally bound to *C-s* and *C-r*. See Chapter 3 [Customization], page 21, to find out how to change the bindings of *C-r or C-s*. For details, see section "Incremental Search" in *The GNU Emacs Manual*

Viper also provides a query replace function that prompts through the Minibuffer. It is invoked by the *Q* key in Vi state.

On a window display, Viper supports mouse search, i.e., you can search for a word by clicking on it. See Section 3.3 [Viper Specials], page 31, for details.

Finally, on a window display, Viper highlights search patterns as it finds them. This is done through what is known as *faces* in Emacs. The variable that controls how search patterns are highlighted is `vip-search-face`. If you don't want any highlighting at all, put

```
(copy-face 'default 'vip-search-face)
```

in '`~/.vip`'. If you want to change how patterns are highlighted, you will have to set the variable `vip-search-face` to some other face, such as `highlight`. If none of the existing faces fits the bill, you may have to create your own. Further details on faces can be found in the Emacs Lisp Manual.

2.7 Abbreviation Facilities

It is possible in Emacs to define abbrevs based on the contents of the buffer. Sophisticated templates can be defined using the Emacs abbreviation facilities. See section "Abbreviations" in *The GNU Emacs Manual*, for details.

Emacs also provides Dynamic Abbreviations. Given a partial word, Emacs will search the buffer to find an extension for this word. For instance, one can type '`Abbreviations`' by typing '`A`', followed by a keystroke that completed the '`A`' to '`Abbreviations`'. Repeated typing will search further back in the buffer, so that one could get '`Abbrevs`' by repeating the keystroke, which appears earlier in the text. Emacs binds this to *ESC /*, so you will have to find a key and bind the function `dabbrev-expand` to that key. Facilities like this make Vi's *:ab* command obsolete.

2.8 Movement and Markers

Viper can be set free from the line–limited movements in Vi, such as *l* refusing to move beyond the line, *ESC* moving one character back, etc. These derive from Ex, which is a line editor. If your '`.vip`' contains

```
(setq vip-ex-style-motion nil)
```

the motion will be a true screen editor motion. One thing you must then watch out for is that it is possible to be on the end-of-line character. The keys *x* and *%* will still work correctly, i.e., as if they were on the last character.

The word-movement commands *w*, *e*, etc., and the associated deletion/yanking commands, *dw*, *yw*, etc., can be made to understand Emacs syntax tables. If the variable `vip-syntax-preference` is set to `'strict-vi` (the default) then the meaning of *word* is the same as in Vi. However, if the value is `'reformed-vi` then the alphanumeric symbols will be those specified by the current Emacs syntax table (which may be different for different major modes) plus the underscore symbol `_`. The user can also specify the value `'emacs`, which would make Viper use exactly the Emacs notion of word. In particular, the underscore may not be part of a word. Finally, if `vip-syntax-preference` is set to `'extended`, Viper words would consist of characters that are classified as alphanumeric *or* as parts of symbols. This is convenient for writing programs and in many other situations.

`vip-syntax-preference` is a local variable, so it can have different values for different major modes. For instance, in programming modes it can have the value `'extended`. In text modes where words contain special characters, such as European (non-English) letters, Cyrillic letters, etc., the value can be `'reformed-vi` or `'emacs`.

Changes to `vip-syntax-preference` should be done in the hooks to various major modes. Furthermore, for these changes to take effect, you should execute `(vip-update-alphanumeric-class)` right after changing the value of `vip-syntax-preference`.

The above discussion of the meaning of Viper's words concerns only Viper's movement commands. In regular expressions, words remain the same as in Emacs. That is, the expressions `\w`, `\>`, `\<`, etc., use Emacs' idea of what is a word, and they don't look into the value of variable `vip-syntax-preference`. This is because Viper doesn't change syntax tables in fear of upsetting the various major modes that set these tables.

Textmarkers in Viper remember the file and the position, so that you can switch files by simply doing *`'a`*. If you set up a regimen for using Textmarkers, this is very useful. Contents of textmarkers can be viewed by *`[marker`*. (Contents of registers can be viewed by *`]register`*).

2.9 New Commands

These commands have no Vi analogs.

`C-x, C-c` *`C-x`* will exit from Vi state and return to Emacs state *temporarily*. If you hit one of these keys, Emacs will believe that you hit that key in Emacs state. For example, if you hit *`C-x`* followed by *`2`*, then the current window will be split into 2 and you will be in Vi state again. Except for novice users, *`C-c`* is also set to temporarily escape to Emacs and execute a command from the current major mode. *`ESC`* will do the same, if you configure ⟨ESC⟩ as Meta by setting `vip-no-multiple-ESC` to nil in '`.vip`'. See Chapter 3 [Customization], page 21. *`C-z`* in Insert state will make Emacs think *`Meta`* has been hit.

`\` Escape to Emacs to execute a single Emacs command. For instance, `\` *`ESC`* will act like a Meta key.

`Q` *`Q`* is for query replace. By default, each string to be replaced is treated as a regular expression. You can use `(setq vip-re-query-replace nil)` in your '`.emacs`' file to turn this off. (For normal searches, *`:se nomagic`* will work. Note that *`:se nomagic`* turns Regexps off completely, unlike Vi).

`v` *`V`* *`C-v`*	These keys are used to visit files. *`v`* will switch to a buffer visiting file whose name can be entered in the Minibuffer. *`V`* is similar, but will use a window different from the current window. *`C-v`* is like *`V`*, except that a new frame (X window) will be used instead of a new Emacs window.
`#`	If followed by a certain character *ch*, it becomes an operator whose argument is the region determined by the motion command that follows (indicated as <move>). Currently, *ch* can be one of *`c`*, *`C`*, *`g`*, *`q`*, and *`s`*. For instance, *`#qr`* will prompt you for a string and then prepend this string to each line in the buffer.
`# c`	Change upper case characters in the region to lower case (`downcase-region`). Emacs command *`M-l`* does the same for words.
`# C`	Change lower case characters in the region to upper case. For instance, *`# C 3 w`* will capitalize 3 words from the current point (`upcase-region`). Emacs command *`M-u`* does the same for words.
`# g`	Execute last keyboard macro for each line in the region (`vip-global-execute`).
`# q`	Insert specified string at the beginning of each line in the region (`vip-quote-region`).
`# s`	Check spelling of words in the region (`spell-region`). The function used for spelling is determined from the variable `vip-spell-function`.
`*`	Call last keyboard macro.
`m .`	Set mark at point and push old mark off the ring

`m<`
`m>`
	Set mark at beginning and end of buffer, respectively.

`m,`	Jump to mark and pop mark off the ring. See section "Mark" in *The GNU Emacs Manual*, for more info.
`] register`	View contents of register
`[ textmarker`	View filename and position of textmarker
`@#` *`@register`* *`@!`*	Begin/end keyboard macro. `@register` has a different meaning when used after a `@#`. See Section 2.4 [Macros and Registers], page 15, for details
`[]`	Go to end of heading.
`g` *<movement command>*	Search buffer for text delimited by movement command. The canonical example is *`gw`* to search for the word under the cursor. See Section 2.6 [Improved Search], page 15, for details.

`_` Meta key when in Vi state.See Section 1.3.2 [Vi State], page 7, for an explanation.

`C-g` and `C-]`
: Quit and Abort Recursive edit. These may be necessary on occasion. See Section 1.3.2 [Vi State], page 7, for a reason.

`C-c g` Hitting `C-c` followed by `g` will display the information on the current buffer. This is the same as hitting `C-g` in Vi, but, as explained above, `C-g` is needed for other purposes in Emacs.

`C-c /` Without a prefix argument, this command toggles case-sensitive/case-insensitive search modes and plain vanilla/regular expression search. With the prefix argument 1, i.e., `1 C-c /`, this toggles case-sensitivity; with the prefix argument 2, toggles plain vanilla search and search using regular expressions. See Section 3.3 [Viper Specials], page 31, for alternative ways to invoke this function.

`M-p` and `M-n`
: In the Minibuffer, these commands navigate through the minibuffer histories, such as the history of search strings, Ex commands, etc.

`C-c M-p` and `C-c M-n`
: In Insert or Replace state, these commands let the user peruse the history of insertion strings used in previous insert or replace commands. Try to hit `C-c M-p` or `C-c M-n` repeatedly and see what happens. See Section 3.3 [Viper Specials], page 31, for more.

In Vi state, these commands let the user peruse the history of Vi-style destructive commands, such as `dw`, `J`, `a`, etc. By repeatedly typing `C-c M-p` or `C-c M-n` you will cycle Viper through the recent history of Vi commands, displaying the commands one by one. Once an appropriate command is found, it can be executed by typing '`.`'.

Since typing `C-c M-p` is tedious, it is more convenient to bind an appropriate function to a function key on the keyboard and use that key. See Section 3.3 [Viper Specials], page 31, for details.

`Ex commands`
: The commands `:args`, `:next`, `:pre` behave differently. `:pwd` exists to get current directory. The commands `:b` and `:B` switch buffers around. See Section 4.4 [File and Buffer Handling], page 51, for details. There are also the new commands `:RelatedFile` and `PreviousRelatedFile` (which abbreviate to `R` and `P`, respectively. See Section 3.3 [Viper Specials], page 31, for details.

Apart from the new commands, many old commands have been enhanced. Most notably, Vi style macros are much more powerful in Viper than in Vi. See Section 3.4 [Vi Macros], page 36, for details.

2.10 Useful Packages

Some Emacs packages are mentioned here as an aid to the new Viper user, to indicate what Viper is capable of. A vast number comes with the standard Emacs distribution, and many more exist on the net and on the archives.

This manual also mentions some Emacs features a new user should know about. The details of these are found in the GNU Emacs Manual.

The features first. For details, look up the Emacs Manual.

‘`Make`’
: Makes and Compiles can be done from the editor. Error messages will be parsed and you can move to the error lines.

‘`Shell`’
: You can talk to Shells from inside the editor. Your entire shell session can be treated as a file.

‘`Mail`’
: Mail can be read from and sent within the editor. Several sophisticated packages exist.

‘`Language Sensitive Editing`’
: Editing modes are written for most computer languages in existence. By controlling indentation, they catch punctuation errors.

The packages, below, represents a drop in the sea of special-purpose packages that come with standard distribution of Emacs 19.

‘`Transparent FTP`’
: `ange-ftp.el` can ftp from the editor to files on other machines transparent to the user.

‘`RCS Interfaces`’
: `vc.el` for doing RCS commands from inside the editor

‘`Directory Editor`’
: `dired.el` for editing contents of directories and for navigating in the file system.

‘`Syntactic Highlighting`’
: `hilit19.el` and `font-lock.el` for automatic highlighting various parts of a buffer using different fonts and colors.

‘`Saving Emacs Configuration`’
: `desktop.el` for saving/restoring configuration on Emacs exit/startup.

‘`Spell Checker`’
: `ispell.el` for spell checking the buffer, words, regions, etc.

‘`File and Buffer Comparison`’
: `ediff.el` for finding differences between files and for applying patches.

Emacs Lisp archives exist on ‘`archive.cis.ohio-state.edu`’ and ‘`wuarchive.wustl.edu`’

3 Customization

Customization can be done in 2 ways.

- Elisp code in a '`.vip`' file in your home directory. Viper loads '`.vip`' just before it does the binding for mode hooks. This is the recommended method.
- Elisp code in your '`.emacs`' file before and after the `(require 'viper)` line. This method is not recommended, unless you are know what you are doing.

Emacs customization is done in Emacs Lisp. For the common cases, examples are provided that you can use directly.

3.1 Rudimentary Changes

An easy way to customize Viper is to change the values of constants used in Viper. Here is the list of the constants used in Viper and their default values. The corresponding :se command is also indicated. (The symbols `t` and `nil` represent "true" and "false" in Lisp).

Viper supports both the abbreviated Vi variable names and their full names. Variable completion is done on full names only. *TAB* and *SPC* complete variable names. Typing '=' will complete the name and then will prompt for a value, if applicable. For instance, *:se auSPC* will complete the command to *:set autoindent*; *:se taSPC* will complete the command and prompt further like this: *:set tabstop =* . However, typing *:se tsSPC* will produce a "No match" message because *ts* is an abbreviation for *tabstop* and Viper supports completion on full names only. However, you can still hit *RET* or *=*, which will complete the command like this: *:set ts =* and Viper will be waiting for you to type a value for the tabstop variable. To get the full list of Vi variables, type *:se SPC TAB*.

`vip-auto-indent nil`
`:se ai (:se autoindent)`
`:se gai (:se global-autoindent)`

If `t`, enable auto indentation. by *RET*, *o* or *O* command.

`vip-auto-indent` is a local variable. To change the value globally, use `setq-default`. It may be useful for certain major modes to have their own values of `vip-auto-indent`. This can be achieved by using `setq` to change the local value of this variable in the hooks to the appropriate major modes.

:se ai changes the value of `vip-auto-indent` in the current buffer only; *:se gai* does the same globally.

`vip-electric-mode t`

If `t`, auto-indentation becomes electric, which means that *RET*, *O*, and *o* indent cursor according to the current major mode. In the future, this variable may control additional electric features.

This is a local variable: `setq` changes the value of this variable in the current buffer only. Use `setq-default` to change the value in all buffers.

`vip-case-fold-search nil`
`:se ic (:se ignorecase)`

If `t`, search ignores cases. This can also be toggled by quickly hitting `/` twice.

`vip-re-search nil`
`:se magic` If `t` then search is reg-exp search, if `nil` then vanilla search. This behavior can also be toggled by quickly hitting `/` trice.

`buffer-read-only`
`:se ro (:se readonly)`

Set current buffer to read only. To change globally put `(setq-default buffer-read-only t)` in your '`.emacs`' file.

`blink-matching-paren t`
`:se sm (:se showmatch)`

Show matching parens by blinking cursor.

`tab-width t (default setting via setq-default)`
`:se ts=value (:se tabstop=value)`
`:se gts=value (:se global-tabstop=value)`

`tab-width` is a local variable that controls the width of the tab stops. To change the value globally, use `setq-default`; for local settings, use `setq`.

The command *`:se ts`* sets the tab width in the current buffer only; it has no effect on other buffers.

The command *`:se gts`* sets tab width globally, for all buffers where the tab is not yet set locally, including the new buffers.

Note that typing *`TAB`* normally doesn't insert the tab, since this key is usually bound to a text-formatting function, `indent-for-tab-command` (which facilitates programming and document writing). Instead, the tab is inserted via the command `vip-insert-tab`, which is bound to *`S-tab`* (shift + tab).

On some non-windowing terminals, Shift doesn't modify the *`TAB`* key, so *`S-tab`* behaves as if it were *`TAB`*. In such a case, you will have to bind `vip-insert-tab` to some other convenient key.

`vip-shift-width 8`
`:se sw=value (:se shiftwidth=value)`

The number of columns shifted by `>` and `<` commands.

`vip-search-wrap-around t`
`:se ws (:se wrapscan)`

If `t`, search wraps around the end/beginning of buffer.

`vip-tags-file-name "TAGS"`

The name of the file used as the tag table.

`vip-re-query-replace nil`

If `t`, use reg-exp replace in query replace.

`vip-want-ctl-h-help nil`

If `t`, *`C-h`* is bound to `help-command`; if `nil`, it is bound to `delete-backward-char`.

`vip-vi-style-in-minibuffer` t
: If `t`, Viper provides a high degree of compatibility with Vi insert mode when you type text in the Minibuffer; if `nil`, typing in the Minibuffer feels like plain Emacs.

`vip-no-multiple-ESC` t
: If you set this to `nil`, you can use *`ESC`* as Meta in Vi state. Normally, this is not necessary, since graphical displays have separate Meta keys (usually on each side of the space bar). On a dumb terminal, Viper sets this variable to `twice`, which is almost like `nil`, except that double *`ESC`* beeps. This, too, lets ESC to be used as a Meta.

`vip-keysequence-delay 140`
: Escape sequences separated by this much delay are interpreted as command, ignoring the special meaning of ESC in VI. The default is suitable for most terminals. However, if your terminal is extremely slow, you might want to increase this slightly. You will know if your terminal is slow if the ESC key sequences emitted by the arrow keys are interpreted as separately typed characters (and thus the arrow keys won't work). Making this value too large will slow you down, so exercise restraint.

`vip-ex-style-motion` t
: Set this to `nil`, if you want *`l,h`* to cross lines, etc. See Section 2.8 [Movement and Markers], page 16, for more info.

`vip-ex-style-editing-in-insert` t
: Set this to to `nil`, if you want *`ESC`* to not move back and *`C-h`* to not stop at the beginning of a line in Insert state.

`vip-always` t
: Leave it to Viper to decide when a buffer must be brought up in Vi state, Insert state, or Emacs state. This heuristics works well in virtually all cases. This option must be set before Viper is loaded or in the '`.vip`' file.

`vip-custom-file-name "~/.vip"`
: Change this if you want. Must be set in '`.emacs`' (not '`.vip`'!) before Viper is loaded. Note that you have to set it as a string inside double quotes.

`vip-spell-function 'ispell-region`
: Function used by the command *`#c<move>`* to spell.

`ex-nontrivial-find-file-function`
: The value of this variable is the function used to find all files that match a wildcard. This is usually done when the user types *`:e`* and specifies a wildcard in the file name (or if the file name contains unusual symbols (e.g., a space). Viper provides two functions for this: one for Unix-like systems (`vip-ex-nontrivial-find-file-unix`) and one for DOS, W95, and NT (`vip-ex-nontrivial-find-file-ms`). If the default function doesn't quite do what you expect or if you prefer to use "fancy" shells, you may have to write your own version of this function and make it into the value of `ex-nontrivial-find-file-function`. Use `vip-`

`ex-nontrivial-find-file-unix` and `vip-ex-nontrivial-find-file-ms` as examples.

`ex-cycle-other-window t`

If `t`, *`:n`* and *`:b`* will cycle through files in another window, if one exists.

`ex-cycle-through-non-files nil`

`:n` does not normally cycle through buffers. Set this to get buffers also.

`vip-automatic-iso-accents nil`

If *`t`*, ISO accents will be turned on in insert/replace Viper states and turned off in Vi state. This is useful for editing text in European languages. This variable is buffer-local. If used, it should be set in the hooks to the appropriate major modes (usually setting it in `text-mode-hook` is enough).

`vip-want-emacs-keys-in-insert`

This is set to `nil` for user levels 1 and 2 and to `t` for user levels 3 and 4. Users who specify level 5 are allowed to set this variable as they please (the default for this level is `t`). If set to `nil`, complete Vi compatibility is provided in Insert state. This is really not recommended, as this precludes you from using language-specific features provided by the major modes.

`vip-want-emacs-keys-in-vi`

This is set to `nil` for user level 1 and to `t` for user levels 2–4. At level 5, users are allowed to set this variable as they please (the default for this level is `t`). If set to `nil`, complete Vi compatibility is provided in Vi command state. Setting this to `nil` is really a bad idea, unless you are a novice, as this precludes the use of language-specific features provided by the major modes.

`vip-keep-point-on-repeat t`

If `t`, point is not moved when the user repeats the previous command by typing '.' This is very useful for doing repeated changes with the `.` key.

`vip-repeat-from-history-key 'f12`

Prefix key used to invoke the macros *`f12 1`* and *`f12 2`* that repeat the second-last and the third-last destructive command. Both these macros are bound (as Viper macros) to `vip-repeat-from-history`, which checks the second key by which it is invoked to see which of the previous commands to invoke. Viper binds *`f12 1`* and *`f12 2`* only, but the user can bind more in '`~/.vip`'. See Section 3.4 [Vi Macros], page 36, for how to do this.

`vip-keep-point-on-undo nil`

If `t`, Viper tries to not move point when undoing commands. Instead, it will briefly move the cursor to the place where change has taken place. However, if the undone piece of text is not seen in window, then point will be moved to the place where the change took place. Set it to `t` and see if you like it better.

`vip-delete-backwards-in-replace nil`

If `t`, DEL key will delete characters while moving the cursor backwards. If `nil`, the cursor will move backwards without deleting anything.

`vip-replace-overlay-face 'vip-replace-overlay-face`

On a graphical display, Viper highlights replacement regions instead of putting a '`$`' at the end. This variable controls the so called *face* used to highlight the region.

By default, `vip-replace-overlay-face` underlines the replacement on monochrome displays and highlights it with color on color displays. If you know something about Emacs faces and don't like how Viper highlights replacement regions, you can change this variable to specify a new face name. (Emacs faces are described in the Emacs Lisp reference.) On a color display, the following customization method is usually most effective:

```
(set-face-foreground vip-replace-overlay-face "DarkSlateBlue")
(set-face-background vip-replace-overlay-face "yellow")
```

For a complete list of colors available to you, evaluate the expression `(x-defined-colors)`. (Type it in the buffer `*scratch*` and then hit the *`C-j`* key.

`vip-replace-overlay-cursor-color "Red"`

Cursor color when it is inside the replacement region. This has effect only on color displays and only when Emacs runs as an X application.

`vip-replace-region-end-delimiter "$"`

A string used to mark the end of replacement regions. It is used only with TTYs or if `vip-use-replace-region-delimiters` is non-nil.

`vip-replace-region-start-delimiter ""`

A string used to mark the beginning of replacement regions. It is used only with TTYs or if `vip-use-replace-region-delimiters` is non-nil.

`vip-use-replace-region-delimiters`

If non-nil, Viper will always use `vip-replace-region-end-delimiter` and `vip-replace-region-start-delimiter` to delimit replacement regions, even on color displays (where this is unnecessary). By default, this variable is non-nil only on TTYs or monochrome displays.

`vip-toggle-key "\C-z"`

Specifies the key used to switch from Emacs to Vi and back. Must be set in '`.vip`' or prior to loading Viper. This variable can't be changed interactively after Viper is loaded.

`vip-ESC-key "\e"`

Specifies the key used to escape from Insert/Replace states to Vi. Must be set in '`.vip`' or prior to loading Viper. This variable cannot be changed interactively after Viper is loaded.

`vip-buffer-search-char nil`

Key used for buffer search. See Section 3.3 [Viper Specials], page 31, for details.

`vip-surrounding-word-function 'vip-surrounding-word`

The value of this variable is a function name that is used to determine what constitutes a word clicked upon by the mouse. This is used by mouse search and insert.

`vip-search-face 'vip-search-face`
: Variable that controls how search patterns are highlighted when they are found.

`vip-vi-state-hook nil`
: List of parameterless functions to be run just after entering the Vi command state.

`vip-insert-state-hook nil`
: Same for Insert state. This hook is also run after entering Replace state.

`vip-replace-state-hook nil`
: List of (parameterless) functions called just after entering Replace state (and after all `vip-insert-state-hook`).

`vip-emacs-state-hook nil`
: List of (parameterless) functions called just after switching from Vi state to Emacs state.

`vip-load-hook nil`
: List of (parameterless) functions called just after loading Viper. This is the last chance to do customization before Viper is up and running.

You can reset some of these constants in Viper with the Ex command *`:set`* (when so indicated in the table). Or you can include a line like this in your '`.vip`' file:

```
(setq vip-case-fold-search t)
```

3.2 Keybindings

Viper lets you define hot keys, i.e., you can associate keyboard keys such as F1, Help, PgDn, etc., with Emacs Lisp functions (that may already exist or that you will write). Each key has a "preferred form" in Emacs. For instance, the Up key's preferred form is [up], the Help key's preferred form is [help], and the Undo key has the preferred form [f14]. You can find out the preferred form of a key by typing *`M-x describe-key-briefly`* and then typing the key you want to know about.

Under X Windows, every keyboard key emits its preferred form, so you can just type

```
(global-set-key [f11] 'calendar)                        ; L1, Stop
(global-set-key [f14] 'undo)                            ; L4, Undo
```

to bind L1 so it will invoke the Emacs Calendar and to bind L4 so it will undo changes. However, on a dumb terminal or in an Xterm window, even the standard arrow keys may not emit the right signals for Emacs to understand. To let Emacs know about those keys, you will have to find out which key sequences they emit by typing *`C-q`* and then the key (you should switch to Emacs state first). Then you can bind those sequences to their preferred forms using `function-key-map` as follows:

```
(cond ((string= (getenv "TERM") "xterm")
       (define-key function-key-map "\e[192z" [f11])    ; L1
       (define-key function-key-map "\e[195z" [f14])    ; L4, Undo
```

The above illustrates how to do this for Xterm. On VT100, you would have to replace "xterm" with "vt100" and also change the key sequences (the same key may emit different sequences on different types of terminals).

The above keys are global, so they are overwritten by the local maps defined by the major modes and by Viper itself. Therefore, if you wish to change a binding set by a major mode or by Viper, read this.

Viper users who wish to specify their own key bindings should be concerned only with the following three keymaps: `vip-vi-global-user-map` for Vi state commands, `vip-insert-global-user-map` for Insert state commands, and `vip-emacs-global-user-map` for Emacs state commands (note: customized bindings for Emacs state made to `vip-emacs-global-user-map` are *not* inherited by Insert state).

For more information on Viper keymaps, see the header of the file '`viper.el`'. If you wish to change a Viper binding, you can use the `define-key` command, to modify `vip-vi-global-user-map`, `vip-insert-global-user-map`, and `vip-emacs-global-user-map`, as explained below. Each of these key maps affects the corresponding Viper state. The keymap `vip-vi-global-user-map` also affects Viper's Replace state.

If you want to bind a key, say *`C-v`*, to the function that scrolls page down and to make *`0`* display information on the current buffer, putting this in '`.vip`' will do the trick in Vi state:

```
(define-key vip-vi-global-user-map "\C-v" 'scroll-down)
```

To set a key globally,

```
(define-key vip-emacs-global-user-map "\C-c m" 'smail)
(define-key vip-vi-global-user-map "0" 'vip-info-on-file)
```

Note, however, that this binding may be overwritten by other keymaps, since the global keymap has the lowest priority. To make sure that nothing will override a binding in Emacs state, you can write this:

```
(define-key vip-emacs-global-user-map "\C-c m" 'smail)
```

To customize the binding for *`C-h`* in Insert state:

```
(define-key vip-insert-global-user-map "\C-h" 'my-del-backwards-function)
```

Each Emacs command key calls some lisp function. If you have enabled the Help, (See Section 3.1 [Rudimentary Changes], page 21) *`C-h k`* will show you the function for each specific key; *`C-h b`* will show all bindings, and *`C-h m`* will provide information on the major mode in effect. If Help is not enabled, you can still get help in Vi state by prefixing the above commands with \, e.g., \ *`C-h k`* (or you can use the Help menu in the menu bar, if Emacs runs under X Windows).

Viper users can also change bindings on a per major mode basis. As with global bindings, this can be done separately for each of the three main Viper states. To this end, Viper provides the function `vip-modify-major-mode`.

To modify keys in Emacs state for `my-favorite-major-mode`, the user needs to create a sparse keymap, say, `my-fancy-map`, bind whatever keys necessary in that keymap, and put

```
(vip-modify-major-mode 'dired-mode 'emacs-state my-fancy-map)
```

in '`~/.vip`'. To do the same in Vi and Insert states, one should use `vi-state` and `insert-state`. Changes in Insert state are also in effect in Replace state. For instance, suppose that the user wants to use *dd* in Vi state under Dired mode to delete files, *u* to unmark files, etc. The following code in '`~/.vip`' will then do the job:

```
(setq my-dired-modifier-map (make-sparse-keymap))
(define-key my-dired-modifier-map "dd" 'dired-flag-file-deletion)
(define-key my-dired-modifier-map "u" 'dired-unmark)
(vip-modify-major-mode 'dired-mode 'vi-state my-dired-modifier-map)
```

A Vi purist may want to modify Emacs state under Dired mode so that *k*, *l*, etc., will move around in directory buffers, as in Vi. Although this is not recommended, as these keys are bound to useful Dired functions, the trick can be accomplished via the following code:

```
(setq my-dired-vi-purist-map (make-sparse-keymap))
(define-key my-dired-vi-purist-map "k" 'vip-previous-line)
(define-key my-dired-vi-purist-map "l" 'vip-forward-char)
(vip-modify-major-mode 'dired-mode 'emacs-state my-dired-vi-purist-map)
```

Similar effect can be achieved by defining Vi keyboard macros using the Ex commands *:map* and *:map!*. The difference is that multi-key Vi macros do not override the keys they are bound to, unless these keys are typed in quick succession. So, with macros, one can use the normal keys alongside with the macros. If per-mode modifications are needed, the user can try both ways and see which one is more convenient. See Section 3.4 [Vi Macros], page 36, for details.

Note: in major modes that come up in *Emacs state* by default, the aforesaid modifications may not take place immediately (but only after the buffer switches to some other Viper state and then back to Emacs state). To avoid this, one should add `vip-change-state-to-emacs` to an appropriate hook of that major mode. (Check the function `vip-set-hooks` in '`viper.el`' for examples.) However, if you have set `vip-always` to `t`, chances are that you won't need to perform the above procedure, because Viper will take care of most useful defaults.

Finally, Viper has a facility that lets the user define per-buffer bindings, i.e., bindings that are in effect in some specific buffers only. Unlike per-mode bindings described above, per-buffer bindings can be defined based on considerations other than the major mode. This is done via the function `vip-add-local-keys`, which lets one specify bindings that should be in effect in the current buffer only and for a specific Viper state. For instance,

```
(vip-add-local-keys 'vi-state '(("ZZ" . TeX-command-master)
                                ("ZQ" . vip-save-kill-buffer)))
```

redefines *ZZ* to invoke `TeX-command-master` in `vi-state` and *ZQ* to save-then-kill the current buffer. These bindings take effect only in the buffer where this command is executed. The typical use of this function is to execute the above expression from within a function that is included in a hook to some major mode. For instance, the above expression could be called from a function, `my-tex-init`, which may be added to `tex-mode-hook` as follows:

```
(add-hook 'tex-mode-hook 'my-tex-init)
```

When TeX mode starts, the hook is executed and the above Lisp expression is evaluated. Then, the bindings for *ZZ* and *ZQ* are changed in Vi command mode for all buffers in TeX mode.

Another useful application is to bind *ZZ* to `send-mail` in the Mail mode buffers (the specifics of this depend on which mail package you are using, `rmail`, `mh-e`, `vm`, etc. For instance, here is how to do this for `mh-e`, the Emacs interface to MH:

```
(defun mh-add-vi-keys ()
  "Set up ZZ for MH-e and XMH."
  (vip-add-local-keys 'vi-state '(("ZZ" . mh-send-letter))))
(add-hook 'mh-letter-mode-hook 'mh-add-vi-keys)
```

You can also use `vip-add-local-keys` to set per buffer bindings in Insert state and Emacs state by passing as a parameter the symbols `'insert-state` and `'emacs-state`, respectively. As with global bindings, customized local bindings done to Emacs state are not inherited by Insert state.

On rare occasions, local keys may be added by mistake. Usually this is done indirectly, by invoking a major mode that adds local keys (e.g., `shell-mode` redefines *RET*). In such a case, exiting the wrong major mode won't rid you from unwanted local keys, since these keys are local to Viper state and the current buffer, not to the major mode. In such situations, the remedy is to type *M-x vip-zap-local-keys*.

So much about Viper-specific bindings. See section "Customization" in *The GNU Emacs Manual*, and the Emacs quick reference card for the general info on key bindings in Emacs.

3.2.1 Packages that Change Keymaps

Viper is designed to coexist with all major and minor modes of Emacs. This means that bindings set by those modes are generally available with Viper (unless you explicitly prohibit them by setting `vip-want-emacs-keys-in-vi` and `vip-want-emacs-keys-in-insert` to `nil`). If `vip-always` is set to `t`, Viper will try to bring each buffer in the a Viper state that is most appropriate for that buffer. Usually, this would be the Vi state, but sometimes it could be the Insert state or the Emacs state.

Some major mode bindings will necessarily be overwritten by Viper. Indeed, in Vi state, most of the 1-character keys are used for Vi-style editing. This usually causes no problems because most packages designed for editing files typically do not bind such keys. Instead, they use key sequences that start with *C-x* and *C-c*. This is why it was so important for us to free up *C-x* and *C-c*. It is common for language-specific major modes to bind *TAB* and *LFD* (the line feed) keys to various formatting functions. This is extremely useful, but may require some getting used to for a Vi user. If you decide that this feature is not for you, you can re-bind these keys as explained earlier (See Chapter 3 [Customization], page 21).

Binding for *TAB* is one of the most unusual aspects of Viper for many novice users. In Emacs, *TAB* is used to format text and programs, and is extremely useful. For instance, hitting *TAB* causes the current line to be re-indented in accordance with the context. In programming, this is very important, since improper automatic indentation would immediately alert the programmer to a possible error. For instance, if a *)* or a " is missing somewhere above the current line, *TAB* is likely to mis-indent the line.

For this reason, Viper doesn't change the standard Emacs binding of *TAB*, thereby sacrificing Vi compatibility (except for users at level 1). Instead, in Viper, the key *S-tab* (shift+ tab) is chosen to emulate Vi's *TAB*.

We should note that on some non-windowing terminals, Shift doesn't modify the *TAB* key, so *S-tab* behaves as if it were *TAB*. In such a case, you will have to bind `vip-insert-tab` to some other convenient key.

Some packages, notably Dired, Gnus, Info, etc., attach special meaning to common keys like *SPC*, *x*, *d*, *v*, and others. This means that Vi command state is inappropriate for working with these packages. Fortunately, these modes operate on read-only buffers and are designed not for editing files, but for special-purpose browsing, reading news, mail, etc., and Vi commands are meaningless in these situations. For this reason, Viper doesn't force Vi state on such major modes. Rather, it brings them in Emacs state. You can switch to Vi state by typing *C-z* if, for instance, you want to do Vi-style search in a buffer (although, usually, incremental search, which is bound to *C-s*, is sufficient in these situations). But you should then switch back to Emacs state if you plan to continue using these major modes productively. You can also switch to Vi temporarily, to execute one 1-character command. This is done by typing *M-C-z* (or *ESC C-z*). This facility cannot execute several complex Vi commands, such as *dw*, *de*, etc.

It is also possible to harness some major modes, even though they may bind common keys to specialized commands. Harnessing can make sense for modes that bind only a small number of common keys. For instance, if `vip-always` is set to `t` in your '`~/.vip`' file, Viper will harness the Shell mode by changing the bindings for *C-m* and *C-d* using `vip-add-local-keys` described in section on customization (See Chapter 3 [Customization], page 21). In general, there is no single recipe for harnessing modes. It can be as simple as adding the function `viper-mode` to a hook associated with the mode, or it can be more complex, as in the case of Shell mode and Emerge. Take a look at `vip-set-hooks` function for some examples.

Conversely, it may be the case that most of the major modes harnessed by `vip-set-hooks` function fit your working style, except one or two cases. In this case, you may still be able to set `vip-always` to `t` and then remove a hook that forces Vi command state. For instance, to unharness `lisp-interaction-mode`, you can put the following line in your '`.emacs`' (not .vip!) file after `(require 'viper)`:

```
(remove-hook 'lisp-interaction-mode-hook 'viper-mode)
```

In some rare cases, some minor modes may override certain essential bindings in Vi command state. This is not really catastrophic because this may happen only in the beginning, when the minor mode kicks in. Typing `M-x viper-mode` will correct the situation. Viper knows about several such minor modes and takes care of them, so that the above trick is usually not necessary. If you find that some minor mode, e.g., `nasty-mode.el` interferes with Viper, putting the following in '`.vip`' should fix the problem:

```
(vip-harness-minor-mode "nasty-mode")
```

The argument to `vip-harness-minor-mode` is the name of the file for the offending minor mode with the suffixes '`.el`' and '`.elc`' removed.

It may be tricky, however, to find out which minor mode is at fault. The only guidance here is to look into the file that defines the minor mode you are suspecting, say `nasty-mode.el`, and see if it has a variable called `nasty-mode-map`. Then check if there is a statement of the form

```
(define-key nasty-mode-map key function)
```

that binds the misbehaving keys. If so, use the above line to harness `nasty-mode`. If your suspicion is wrong, no harm is done if you harness a minor mode that doesn't need to be harnessed.

3.3 Viper Specials

Viper extends Vi with a number of useful features. This includes various search functions, histories of search strings, Ex commands, insertions, and Vi's destructive commands. In addition, Viper supports file name completion and history, completion of Ex commands and variables, and many other features. Some of these features are explained in detail elsewhere in this document. Other features are explained here.

`(vip-buffer-search-enable)`

`vip-buffer-search-char nil`

Enable buffer search. Explicit call to `vip-buffer-search-enable` sets `vip-buffer-search-char` to *g*. Alternatively, the user can set `vip-buffer-search-char` in '`.vip`' to a key sequence to be used for buffer search. There is no need to call `vip-buffer-search-enable` in that case.

`vip-toggle-search-style`

This function, bound to *C-c /*, lets one toggle case-sensitive and case-insensitive search, and also switch between plain vanilla search and search via regular expressions. Without the prefix argument, the user is asked which mode to toggle. With prefix argument 1, this toggles case-sensitivity. With prefix argument 2, regular expression/vanilla search will be toggled.

However, we found that the most convenient way to toggle these options is to bind a Vi macro to bind *//* to toggles case sensitivity and to *///* to toggles vanilla search. Thus, quickly hitting */* twice will switch Viper from case sensitive search to case-insensitive. Repeating this once again will restore the original state. Likewise, quickly hitting */* three times will switch you from vanilla-style search to search via regular expressions. If you hit something other than */* after the first */* or if the second */* doesn't follow quickly enough, then Viper will issue the usual prompt */* and will wait for input, as usual in Vi. If you don't like this behavior, you can "unrecord" these macros in your '`~/.vip`' file. For instance, if u don't like the above feature, put this in '`~/.vip`':

```
(vip-unrecord-kbd-macro "//" 'vi-state)
(vip-unrecord-kbd-macro "///" 'vi-state)
```

See Section 3.4 [Vi Macros], page 36, for more information on Vi macros.

`vip-heading-start`

`vip-heading-end`

Regular Expressions for *[[* and *]]*. Note that Emacs defines Regexps for paragraphs and sentences. See section "Paragraphs and Sentences" in *The GNU Emacs Manual*, for details.

`M-x vip-set-expert-level`

Change your user level interactively.

`vip-smart-suffix-list '("" "tex" "c" "cc" "el" "p")`

Viper supports Emacs-style file completion when it prompts the user for a file name. However, in many cases, the same directory may contain files with identical prefix but different suffixes, e.g., prog.c, prog.o, paper.tex, paper.dvi. In such cases, completion will stop at the '.'. If the above variable is a list of strings representing suffixes, Viper will try these suffixes in the order listed and will check if the corresponding file exists.

For instance, if completion stopped at 'paper.' and the user typed RET, then Viper will check if the files 'paper.', 'paper.tex', 'paper.c', etc., exist. It will take the first such file. If no file exists, Viper will give a chance to complete the file name by typing the appropriate suffix. If 'paper.' was the intended file name, hitting return will accept it.

To turn this feature off, set the above variable to `nil`.

`vip-insertion-ring-size` 14

Viper remembers what was previously inserted in Insert and Replace states. Several such recent insertions are kept in a special ring of strings of size `vip-insertion-ring-size`. If you enter Insert or Replace state you can reinsert strings from this ring by typing *C-c M-p* or *C-c M-n*. The former will search the ring in the direction of older insertions, and the latter will search in the direction of newer insertions. Hitting *C-c M-p* or *C-c M-n* in succession will undo the previous insertion from the ring and insert the next item on the ring. If a larger ring size is needed, change the value of the above variable in the '`~/.vip`' file.

Since typing these sequences of keys may be tedious, it is suggested that the user should bind a function key, such as *f31*, as follows:

```
(define-key vip-insert-global-user-map [f31]
            'vip-insert-prev-from-insertion-ring)
```

This binds *f31* (which is usually *R11* on a Sun workstation) to the function that inserts the previous string in the insertion history. To rotate the history in the opposite direction, you can either bind an unused key to `vip-insert-next-from-insertion-ring` or hit any digit (1 to 9) then *f31*.

One should not bind the above functions to *M-p* or *M-n*, since this will interfere with the Minibuffer histories and, possibly, other major modes.

`vip-command-ring-size` 14

Viper keeps track of the recent history of destructive commands, such as *dw*, *i*, etc. In Vi state, the most recent command can be re-executed by hitting '`.`', as in Vi. However, repeated typing *C-c M-p* will cause Viper to show the previous destructive commands in the minibuffer. Subsequent hitting '`.`' will execute the command that was displayed last. The key *C-c M-n* will cycle through the command history in the opposite direction. Since typing *C-c M-p* may be tedious, it is more convenient to bind an appropriate function to an unused function key on the keyboard and use that key. For instance, the following

```
(define-key vip-vi-global-user-map [f31]
            'vip-prev-destructive-command)
```

binds the key *`f31`* (which is usually *`R11`* on a Sun workstation) to the function that searches the command history in the direction of older commands. To search in the opposite direction, you can either bind an unused key to `vip-next-destructive-command` or hit any digit (1 to 9) then *`f31`*.

One should not bind the above functions to *`M-p`* or *`M-n`*, since this will interfere with the Minibuffer histories and, possibly, other major modes.

`vip-minibuffer-vi-face 'vip-minibuffer-vi-face`

`vip-minibuffer-insert-face 'vip-minibuffer-insert-face`

`vip-minibuffer-emacs-face 'vip-minibuffer-emacs-face`

These faces control the appearance of the minibuffer text in the corresponding Viper states. For heavy-duty customization, consult the Lisp Reference to Emacs. You can also take a look how these faces are defined in '`viper.el`'.

However, on a color workstation, the following method usually suffices:

```
(set-face-foreground vip-minibuffer-vi-face "blue")
(set-face-background vip-minibuffer-emacs-face "orchid")
```

This will make a blue foreground in the Minibuffer when it is in Vi state; its background will turn to orchid when it switches to Emacs state.

Note that only the text you type in is affected by the above faces. Prompts and Minibuffer messages are not affected.

Purists who do not like adornments in the minibuffer can always zap them by putting

```
(copy-face 'default 'vip-minibuffer-vi-face)
(copy-face 'default 'vip-minibuffer-insert-face)
(copy-face 'default 'vip-minibuffer-emacs-face)
```

in the '`~/.vip`' file. However, in that case, the user will not have any indication of the current Viper state in the minibuffer. (This is important if the user accidentally switches to another Viper state by typing *`ESC`* or *`C-z`*.

Viper provides some support for multi-file documents and programs. If a document consists of several files we can designate one of them as a master and put the following at the end of that file:

```
;;; Local Variables:
;;; eval: (vip-setup-master-buffer "file1" "file2" "file3" "file5" "file5")
;;; End:
```

where `file1` to `file5` are names of files related to the master file. Next time, when the master file is visited, the command `vip-setup-master-buffer` will be evaluated and the above files will be associated with the master file. Then, the new Ex command *`:RelatedFile`* (abbr. *`:R`*) will display files 1 to 5 one after another, so you can edit them. If a file is not in any Emacs buffer, it will be visited. The command *`PreviousRelatedFile`* (abbr., *`:P`*) goes through the file list in the opposite direction.

These commands are akin to *`:n`* and *`:N`*, but they allow the user to focus on relevant files only.

Note that only the master file needs to have the aforementioned block of commands. Also, ";;;" above can be replaced by some other markers. Semicolon is good for Lisp programs, since it is considered a comment designator there. For LaTeX, this could be `"%%%"`, and for C the above block should be commented out.

Even though these commands are sometimes useful, they are no substitute for the powerful *tag table* facility of Emacs. Viper's *`:tag`* command in a primitive interface to Emacs tags. See section "Tags" in *The Gnu Emacs Manual*, for more information on tags.

The following two commands are normally bound to a mouse click and are part of Viper. They work only if Emacs runs as an application under X Windows (or under some other window system for which a port of Emacs 19 is available). Clicking the mouse when Emacs is invoked in an Xterm window (using `emacs -nw`) will do no good.

`M-S-mouse-1 (Emacs)`

`meta shift button1up (XEmacs)`

Holding Meta and Shift while clicking mouse button 1 will initiate search for a region under the mouse pointer (defined below). This command can take a prefix argument, which indicates the occurrence of the pattern to search for.

Note: Viper binds this mouse action only if it is not already bound to something else. If you want to use this feature and *`M-S-mouse-1`* is already used for something else, you can rebind mouse-search as, for example, in the following example:

```
(global-set-key [M-mouse-1] 'vip-mouse-click-search-word)
(global-set-key [M-down-mouse-1] 'vip-mouse-catch-frame-switch)
```

This would bind mouse search to the action invoked by pressing the Meta key and clicking mouse button 1. Note: if `vip-mouse-click-search-word` is bound to an action, then `vip-mouse-catch-frame-switch` must be bound to a down-action, as shown in the above example.

In XEmacs, you can change bindings as follows:

```
(global-set-key [(meta control button1up)]
                'vip-mouse-click-search-word)
(global-set-key [(meta control button1)]
                'vip-mouse-catch-frame-switch)
```

if, say, you prefer to hold both meta and control while clicking.

Like in Emacs, there are special rules for binding these functions: the first must be bound to a button-up event while the second must be bound to a button-event (which is XEmacs' equivalent of a down-mouse event). Also, in Emacs, the double-click and triple-click actions for the same button (`double-S-mouse-1`, etc., if the above default binding is used) should not be bound (or it should be bound to the same function, `vip-mouse-click-search-word`).

The region that is chosen as a pattern to search for is determined as follows. If search is invoked via a single click, Viper chooses the region that lies between the beginning of the "word" under the pointer ("word" is understood in Vi sense) and the end of that word. The only difference with Vi's words is that in Lisp major modes '-' is considered an alphanumeric symbol. This is done for the convenience of working with Lisp symbols, which often have an '-' in them. Also, if you click on a non-alphanumeric character that is not a word separator (in Vi sense) then this character will also be considered alphanumeric, provided that it is adjacent (from either side) to an alphanumeric character. This useful feature gives added control over the patterns selected by the mouse click.

On a double-click, the region is determined by the beginning of the current Vi's "Word" (i.e., the largest non-separator chunk of text) and the End of that "Word" (as determined by the *`E`* command).

On a triple-click, the region consists of the entire line where the click occurred with all leading and trailing spaces and tabs removed.

`M-S-mouse-2 (Emacs)`

`meta shift button2up (XEmacs)`

Holding Meta and Shift while clicking mouse button 2 will insert the region surrounding the mouse pointer. The rules defining this region are the same as for mouse-search. This command takes an optional prefix argument, which indicates how many such regions to snarf from the buffer and insert. (In case of a triple-click, the prefix argument is ignored.)

Note: Viper binds this mouse action only if it not already bound to something else. If you want to use this feature and *`S-mouse-2`* is already used for something else, you can rebind mouse-insert as follows:

```
(global-set-key [M-mouse-2] 'vip-mouse-click-insert-word)
(global-set-key [M-down-mouse-2] 'vip-mouse-catch-frame-switch)
```

In XEmacs, you can change the bindings as follows:

```
(global-set-key [(meta control button2up)]
                'vip-mouse-click-insert-word)
(global-set-key [(meta control button2)]
                'vip-mouse-catch-frame-switch)
```

`vip-multiclick-timeout`

This variable controls the rate at which double-clicking must occur for the purpose of mouse search and mouse insert. By default, this is set to `double-click-time` in Emacs and to `mouse-track-multi-click-time` milliseconds in XEmacs.

Note: The above functions search and insert in the selected window of the latest active frame. This means that you can click in another window or another frame and have search or insertion done in the frame and window you just left. This lets one use these functions in a multi-frame configuration. However, this may require some getting used to. For instance, if you are typing in a frame, A, and then move the mouse to frame B and click to invoke mouse search, search (or insertion) will be performed in frame A. To perform search/insertion in frame B, you will first have to shift focus there, which doesn't happen until you type a character or perform some other action in frame B—mouse search doesn't shift focus (in XEmacs, to shift focus to frame B, you will have to select this frame with a mouse, by clicking.

If you decide that you don't like the above feature and always want search/insertion be performed in the frame where the click occurs, don't bind (and unbind, if necessary) `vip-mouse-catch-frame-switch` from the mouse event it is bound to.

Mouse search is integrated with Vi-style search, so you can repeat it with *`n`* and *`N`*. It should be also noted that, while case-sensitivity of search in Viper is controlled by the variable `vip-case-fold-search`, the case of mouse search is controlled by the Emacs variable `case-fold-search`, which may be set differently from `vip-case-fold-search`. Therefore, case-sensitivity of mouse search may be different from that of the usual Vi-style search.

Finally, if the way Viper determines the word to be searched for or to be inserted is not what you want, there is a variable, `vip-surrounding-word-function`, which can be changed to indicate another function for snarfing words out of the buffer. The catch is that you will then have to write such a function and make it known to your Emacs. The function `vip-surrounding-word` in '`viper.el`' can be used as a guiding example.

3.4 Vi Macros

Viper supports much enhanced Vi-style macros and also facilitates the use of Emacs-style macros. To define a temporary macro, it is generally more convenient to use Emacs keyboard macro facility. Emacs keyboard macros are usually defined anonymously, and the latest macro can be executed by typing *`C-x e`* (or *`*`*, if Viper is in Vi state). If you need to use several temporary macros, Viper lets you save them to a register (a lowercase letter); such macros can then be executed by typing `@a` in Vi state (if a macro was previously saved in register `a`). See Section 2.4 [Macros and Registers], page 15, for details.

If, however, you need to use a macro regularly, it must be given a permanent name and saved. Emacs manual explains how to do this, but invocation of named Emacs macros is quite different from Vi's. First, invocation of permanent Emacs macros takes time because of the extra keys. Second, binding such macros to function keys, for fast access, hogs valuable real estate on the keyboard.

Vi-style macros are better in that respect, since Vi lets the user overload the meaning of key sequences: keys typed in fast succession are treated specially, if this key sequence is bound to a macro.

Viper provides keyboard macros through the usual Ex commands, *`:map`* and *`:map!`*. Vi-style macros are much more powerful in Viper than they are in the original Vi and in other emulators. This is because Viper implements an enhanced vi-style interface to the powerful Emacs keyboard macro facility.

First, any Emacs command can be executed while defining a macro, not just the Vi commands. In particular, the user can invoke Emacs commands via *`M-x command-name`* or by pressing various function keys on the keyboard. One can even use the mouse, although this is usually not useful and is not recommended (and macros defined with the use of the mouse cannot be saved in command history and in the startup file, for future use).

Macros defined by mixing Vi and Emacs commands are represented as vectors. So, don't be confused when you see one (usually through the history of Ex commands). For instance, if *`gg`* is defined by typing *`l`*, the up-arrow key and *`M-x next-line`*, its definition will look as follows in Emacs (in XEmacs, it looks slightly different, see below):

```
[l up (meta x) n e x t - l i n e return]
```

Second, Viper macros are defined in a WYSIWYG style. This means that commands are executed as you type them, so you can see precisely what is being defined. Third, macros can be bound to arbitrary sequences of keys, not just to printable keys. For instance, one can define a macro that will be invoked by hitting `f3` then `f2` function keys. (The keys `delete` and `backspace` are excluded; also, a macro invocation sequence can't start with ESC. Some other keys, such as `f1` and `help`, can't be bound to macros under Emacs (not XEmacs), since they are bound in `key-translation-map`, which overrides any other binding the user gives to keys. In general, keys that have a binding in `key-translation-map` can't be bound to a macro.)

Fourth, in Viper, one can define macros that are specific to a given buffer, a given major mode, or macros that are defined for all buffers. In fact, the same macro name can have several different definitions: one global, several definitions for various major modes, and definitions for various specific buffers. Buffer-specific definitions override mode-specific definitions, which, in turn, override global definitions.

As if all that is not enough, Viper (through its interface to Emacs macros) lets the user define keyboard macros that ask for confirmation or even prompt the user for input and then continue. To do this, one should type `C-x q` (for confirmation) or `C-u C-x q` (for prompt). For details, see section "Customization" in *The GNU Emacs Manual*

When the user finishes defining a macro (which is done by typing `C-x)` — a departure from Vi), you will be asked whether you want this macro to be global, mode-specific, or buffer-specific. You will also be given a chance to save the macro in your '`~/.vip`' file. This is the easiest way to save a macro and make it permanently available. If you work your startup files with bare hands, here is how Viper saves the above macro so that it will be available in Viper's Insert state (and Replace state) in buffer `my-buf` only:

```
(vip-record-kbd-macro "gg" 'insert-state
       [l up (meta x) n e x t - l i n e return]
       "my-buf")
```

To do the same for Vi state and all buffers with the major mode `cc-mode`, use:

```
(vip-record-kbd-macro "gg" 'vi-state
       [l up (meta x) n e x t - l i n e return]
       'cc-mode)
```

Both macro names and macro definitions are vectors of symbols that denote keys on the keyboard. Some keys, like `\`, , or digit-keys must be escaped with a backslash. Modified keys are represented as lists. For instance, holding Meta and Control and pressing `f4` is represented as `(control meta f4)`. If all members of a vectors are printable characters (or sequences, such as `\e`, `\t`, for ESC and TAB), then they can also be represented as strings:

```
(vip-record-kbd-macro "aa" 'vi-state  "aaa\e"  "my-buffer")
```

Thus, typing `aa` fast in Vi state will switch Viper to Insert state (due to the first `a`), insert `aa`, and then it will switch back to Vi state. All this will take effect only in the buffer named `my-buffer`.

Note that the last argument to `vip-record-kbd-macro` must be either a string (a buffer name), a symbol representing a major mode, or `t`; the latter says that the macro is to be defined for all buffers (which is how macros are defined in original Vi).

For convenience, Viper also lets you define Vi-style macros in its Emacs state. There is no Ex command, like `:map` and `:map!` for doing this, but the user can include such a macro in the '`~/.vip`' file. The only thing is that the `vip-record-kbd-macro` command should specify `'emacs-state` instead of `'vi-state` or `'insert-state`.

The user can get rid of a macro either by using the Ex commands *`:unmap`* and *`:unmap!`* or by issuing a call to `vip-unrecord-kbd-macro`. The latter is more powerful, since it can delete macros even in `'emacs-state`. However, `vip-unrecord-kbd-macro` is usually needed only when the user needs to get rid of the macros that are already predefined in Viper. The syntax is:

```
(vip-unrecord-kbd-macro macro state)
```

The second argument must be `'vi-state`, `'insert-state`, or `'emacs-state`. The first argument is a name of a macro. To avoid mistakes in specifying names of existing macros, type *`M-x vip-describe-kbd-macros`* and use a name from the list displayed by this command.

If an error occurs during macro definition, Emacs aborts the process, and it must be repeated. This is analogous to Vi, except that in Vi the user doesn't know there is an error until the macro is actually run. All that means that in order for a definition to be successful, the user must do some simple planning of the process in advance, to avoid errors. For instance, if you want to map *`gg`* to *`llll`* in Vi state, you must make sure that there is enough room on the current line. Since *`l`* moves the cursor forward, it may signal an error on reaching the end of line, which will abort the definition.

These precautions are necessary only when defining macros; they will help avoid the need to redo the job. When macros are actually run, an error during the execution will simply terminate the current execution (but the macro will remain mapped).

A macro name can be a string of characters or a vector of keys. The latter makes it possible to define macros bound to, say, double-hits on a function key, such as *`up`* or *`f13`*. This is very useful if you run out of function keys on your keyboard; it makes Viper macro facility a *keyboard doubler*, so to speak.

Elsewhere (See Section 3.2 [Keybindings], page 26, for details), we review the standard Emacs mechanism for binding function keys to commands. For instance,

```
(global-set-key [f13] 'repeat-complex-command)
```

binds the key f13 to the Emacs function that repeats the last minibuffer command. Under Viper, however, you may still use this key for additional purposes, if you bind, say, a double-hitting action for that key to some other function. Emacs doesn't allow the user to do that, but Viper does this through its keyboard macro facility. To do this, type *`:map`* first. When you are asked to enter a macro name, hit f13 twice, followed by RET or SPC.

Emacs will now start the mapping process by actually executing Vi and Emacs commands, so that you could see what will happen each time the macro is executed. Suppose now we wanted to bind the key sequence *`f13 f13`* to the command `eval-last-sexp`. To accomplish this, we can type *`M-x eval-last-sexp`* followed by *`C-x )`*. If you answer positively to Viper's offer to save this macro in '`~/.vip`' for future uses, the following will be inserted in that file:

```
(vip-record-kbd-macro [f16 f16] 'vi-state
         [(meta x) e v a l - l a s t - s e x p]
         'lisp-interaction-mode)
```

To illustrate the above point, Viper provides two canned macros, which, by default, are bound to *`[f12 \1]`* and *`[f12 \2]`* (invoked by typing *`f12`* then *`1`* and *`2`*, respectively). These macros are useful shortcuts to Viper's command ring history. The first macro will execute the second-last destructive command (the last one is executed by `.`, as usual). The second macro executes the third-last command.

If you need to go deeper into the command history, you will have to use other commands, as described earlier in this section; or you can bind, say, *`f12 \3`* like this:

```
(vip-record-kbd-macro [f12 \3] 'vi-state
                      [(meta x) r e p e a t - f r o m - h i s t o r y]
                      t)
```

Note that even though the macro uses the function key *`f12`*, the key is actually free and can still be bound to some Emacs function via `define-key` or `global-set-key`.

Viper allows the user to define macro names that are prefixes of other macros. For instance, one can define *`[[`* and *`[[[[`* to be macros. If you type the exact sequence of such keys and then pause, Viper will execute the right macro. However, if you don't pause and, say, type *`[[[[text`* then the conflict is resolved as follows. If only one of the key sequences, *`[[`* or *`[[[[`* has a definition applicable to the current buffer, then, in fact, there is no conflict and the right macro will be chosen. If both have applicable definitions, then the first one found will be executed. Usually this is the macro with a shorter name. So, in our case, *`[[[[text`* will cause the macro *`[[`* to be executed twice and then the remaining keys, *`t e x t`*, will be processed.

When defining macros using *`:map`* or *`:map!`*, the user enters the actually keys to be used to invoke the macro. For instance, you should hit the actual key *`f6`* if it is to be part of a macro name; you do *not* write 'f 6'. When entering keys, Viper displays them as strings or vectors (e.g., "abc" or [f6 f7 a]). The same holds for unmapping. Hitting TAB while typing a macro name in the *`:unmap`* or *`:unmap!`* command will cause name completion. Completions are displayed as strings or vectors. However, as before, you don't actually type "`"`", "[", or "]" that appear in the completions. These are meta-symbols that indicate whether the corresponding macro name is a vector or a string.

One last difference from Vi: Vi-style keyboard macros cannot be defined in terms of other Vi-style keyboard macros (but named Emacs macros are OK). More precisely, while defining or executing a macro, the special meaning of key sequences (as Vi macros) is ignored. This is because it is all too easy to create an infinite loop in this way. Since Viper macros are much more powerful than Vi's it is impossible to detect such loops. In practice, this is not really a limitation but, rather, a feature.

We should also note that Vi macros are disabled in the Minibuffer, which helps keep some potential troubles away.

The rate at which the user must type keys in order for them to be recognized as a timeout macro is controlled by the variable `vip-fast-keyseq-timeout`, which defaults to 200 milliseconds.

For the most part, Viper macros defined in '`~/.vip`' can be shared between Emacs, XEmacs, and X and TTY modes. However, macros defined via function keys may need separate definitions when XEmacs and Emacs have different names for the same keyboard key. For instance, the 'Page Up' key may be known in Emacs as *`prior`* and in XEmacs as *`pgup`*. The problem with TTY may be that the function keys there generate sequences of events instead of a single event (as under a window system). Both Emacs and XEmacs mape some of these sequences back to the logical keys (e.g., the sequences generated by the arrow keys are mapped to `up`, `left`, etc.). However, not all function keys are mapped in this way. Macros that are bound to key sequences that contain such unmapped function keys have to be redefined for TTY's (and possibly for every type of TTY you may be using). To do this, start Emacs on an appropriate TTY device and define the macro using `:map`, as usual.

Finally, Viper provides a function that conveniently displays all macros currently defined. To see all macros along with their definitions, type `M-x vip-describe-kbd-macros`.

4 Commands

This section is a semi-automatically bowdlerized version of the Vi reference created by ‘`maart@cs.vu.nl`’ and others. It can be found on the Vi archives. Very little has been changed for Viper.

4.1 Groundwork

The VI command set is based on the idea of combining motion commands with other commands. The motion command is used as a text region specifier for other commands. We classify motion commands into *point commands* and *line commands*.

The point commands are:

`h`, `l`, `0`, `$`, `w`, `W`, `b`, `B`, `e`, `E`, `(`, `)`, `/`, `?`, `‘`, `f`, `F`, `t`, `T`, `%`, `;`, `,`, `^`

The line commands are:

`j`, `k`, `+`, `-`, `H`, `M`, `L`, `{`, `}`, `G`, `’`, `[[`, `]]`, `[]`

Text Deletion Commands (See Section 4.2.5 [Deleting Text], page 47), Change commands (See Section 4.2.6 [Changing Text], page 47), even Shell Commands (See Section 4.6 [Shell Commands], page 53) use these commands to describe a region of text to operate on.

Viper adds two region descriptors, `r` and `R`. These describe the Emacs regions (See Section 2.1 [Basics], page 13), but they are not movement commands.

The command description uses angle brackets ‘`<>`’ to indicate metasyntactic variables, since the normal conventions of using simple text can be confusing with Viper where the commands themselves are characters. Watch out where `<` shift commands and `<count>` are mentioned together!!!

‘`<move>`’ refers to the above movement commands, and ‘`<a-z>`’ refers to registers or textmarkers from ‘`a`’ to ‘`z`’. Note that the ‘`<move>`’ is described by full move commands, that is to say they will take counts, and otherwise behave like normal move commands. ‘`<address>`’ refers to Ex line addresses, which include

`. <No address>`	Current line
`.+n .-n`	Add or subtract for current line
`number`	Actual line number, use `.=` to get the line number
`’<a-z>`	Textmarker
`$`	Last line
`x,y`	Where x and y are one of the above
`%`	For the whole file, same as (1,$).
`/pat/`	
`?pat?`	Next or previous line with pattern pat

Note that '%' is used in Ex commands to mean current file. If you want a '%' in your command, it must be escaped as '\%'. Similarly, '#' expands to the previous file. The previous file is the first file in *:args* listing. This defaults to previous window in the VI sense if you have one window only.

Others like '`<args> -- arguments`', '`<cmd> -- command`' etc. should be fairly obvious.

Common characters referred to include:

<sp>	Space
<ht>	Tab
<lf>	Linefeed
<esc>	Escape
<cr>	Return, Enter

We also use '`word`' for alphanumeric/non-alphanumeric words, and '`WORD`' for whitespace delimited words. '`char`' refers to any ASCII character, '`CHAR`' to non-whitespace character. Brackets '`[]`' indicate optional parameters; '`<count>`' also optional, usually defaulting to 1. Brackets are elided for '`<count>`' to eschew obfuscation.

Viper's idea of Vi's words is slightly different from Vi. First, Viper words understand Emacs symbol tables. Therefore, all symbols declared to be alphanumeric in a symbol table can automatically be made part of the Viper word. This is useful when, for instance, editing text containing European, Cyrillic, etc., letters.

Second, Viper lets you depart from Vi's idea of a word by changing the value of `vip-syntax-preference`. By default, this variable is set to `'strict-vi`, which means that alphanumeric symbols are exactly as in Vi. However, if the value is `'reformed-vi` then alphanumeric symbols will be those specified by the current Emacs syntax table (which may be different for different major modes) plus the underscore symbol `_`. The user can also specify the value `'emacs`, which would make Viper use exactly the Emacs notion of word. In particular, the underscore may not be part of a word. Finally, if `vip-syntax-preference` is set to `'extended`, Viper words would consist of characters that are classified as alphanumeric *or* as parts of symbols. This is convenient for writing programs and in many other situations.

`vip-syntax-preference` is a local variable, so it can have different values for different major modes. For instance, in programming modes it can have the value `'extended`. In text modes where words contain special characters, such as European (non-English) letters, Cyrillic letters, etc., the value can be `'reformed-vi` or `'emacs`.

Changes to `vip-syntax-preference` should be done in the hooks to various major modes. Furthermore, for these changes to take effect, you should execute `(vip-update-alphanumeric-class)` right after changing the value of `vip-syntax-preference`.

The above discussion concerns only the movement commands. In regular expressions, words remain the same as in Emacs. That is, the expressions `\w`, `\>`, `\<`, etc., use Emacs' idea of what is a word, and they don't look into the value of variable `vip-syntax-preference`. This is because Viper doesn't change syntax tables in order to not thwart the various major modes that set these tables.

The usual Emacs convention is used to indicate Control Characters, i.e C-h for Control-h. *Do not confuse this to mean the separate characters C - h!!!* The ^ is itself, never used to indicate a Control character.

4.2 Text Handling

4.2.1 Move Commands

`<count> h C-h`	<count> chars to the left.
`<count> j <lf> C-n`	<count> lines downward.
`<count> l <sp>`	<count> chars to the right.
`<count> k C-p`	<count> lines upward.
`<count> $`	To the end of line <count> from the cursor.
`<count> ^`	To the first CHAR <count> - 1 lines lower.
`<count> -`	To the first CHAR <count> lines higher.
`<count> + <cr>`	To the first CHAR <count> lines lower.
`0`	To the first char of the line.
`<count> |`	To column <count>
`<count> f<char>`	<count> <char>s to the right (find).
`<count> t<char>`	Till before <count> <char>s to the right.
`<count> F<char>`	<count> <char>s to the left.
`<count> T<char>`	Till after <count> <char>s to the left.
`<count> ;`	Repeat latest `f t F T` <count> times.
`<count> ,`	Repeat latest `f t F T` <count> times in opposite direction.
`<count> w`	<count> words forward.
`<count> W`	<count> WORDS forward.
`<count> b`	<count> words backward.

<count> B	<count> WORDS backward.
<count> e	To the end of word <count> forward.
<count> E	To the end of WORD <count> forward.
<count> G	Go to line <count> (default end-of-file).
<count> H	To line <count> from top of the screen (home).
<count> L	To line <count> from bottom of the screen (last).
M	To the middle line of the screen.
<count>)	<count> sentences forward.
<count> (	<count> sentences backward.
<count> }	<count> paragraphs forward.
<count> {	<count> paragraphs backward.
<count>]]	To the <count>th heading.
<count> [[	To the <count>th previous heading.
<count> []	To the end of <count>th heading.
m<a-z>	Mark the cursor position with a letter.
‘<a-z>	To the mark.
’<a-z>	To the first CHAR of the line with the mark.
[<a-z>	Show contents of textmarker.
]<a-z>	Show contents of register.
‘‘	To the cursor position before the latest absolute jump (of which are examples */* and *G*).
’’	To the first CHAR of the line on which the cursor was placed before the latest absolute jump.
<count> /<string>	To the <count>th occurrence of <string>.
<count> /<cr>	To the <count>th occurrence of <string> from previous */ or ?*.
<count> ?<string>	To the <count>th previous occurrence of <string>.
<count> ?<cr>	To the <count>th previous occurrence of <string> from previous *? or /*.
n	Repeat latest */ ?* (next).

N	Repeat latest search in opposite direction.
C-c /	Without a prefix argument, this command toggles case-sensitive/case-insensitive search modes and plain vanilla/regular expression search. With the prefix argument 1, i.e., *1 C-c /*, this toggles case-sensitivity; with the prefix argument 2, toggles plain vanilla search and search using regular expressions. See Section 3.3 [Viper Specials], page 31, for alternative ways to invoke this function.
%	Find the next bracket and go to its match.

4.2.2 Marking

Emacs mark is referred to in the region specifiers *r* and *R*. See Section 1.1 [Emacs Preliminaries], page 3 and see Section 2.1 [Basics], page 13 for explanation. Also see section "Mark" in *The GNU Emacs manual*, for an explanation of the Emacs mark ring.

m<a-z>	Mark the current file and position with the specified letter.
m .	Set the Emacs mark (See Section 1.1 [Emacs Preliminaries], page 3) at point.
m <	Set the Emacs mark at beginning of buffer.
m >	Set the Emacs mark at end of buffer.
m ,	Jump to the Emacs mark.
:mark <char>	Mark position with text marker named <char>. This is an Ex command.
:k <char>	Same as *:mark*.
‘‘	Exchange point and mark.
’’	Exchange point and mark and go to the first CHAR on line.
’<a-z>	Go to specified Viper mark.
	Go to specified Viper mark and go to the first CHAR on line.

4.2.3 Appending Text

See Section 4.7 [Options], page 54 to see how to change tab and shiftwidth size. See the GNU Emacs manual, or try *C-ha tabs* (If you have turned Emacs help on). Check out the variable `indent-tabs-mode` to put in just spaces. Also see options for word-wrap.

<count> a	<count> times after the cursor.
<count> A	<count> times at the end of line.
<count> i	<count> times before the cursor (insert).
<count> I	<count> times before the first CHAR of the line

<count> o	On a new line below the current (open). The count is only useful on a slow terminal.
<count> O	On a new line above the current. The count is only useful on a slow terminal.
<count> ><move>	Shift the lines described by <count><move> one shiftwidth to the right (layout!).
<count> >>	Shift <count> lines one shiftwidth to the right.
<count> ["<a-z1-9>]p	Put the contents of the (default undo) buffer <count> times after the cursor. The register will be automatically downcased.
<count> ["<a-z1-9>]P	Put the contents of the (default undo) buffer <count> times before the cursor. The register will
[<a-z>	Show contents of textmarker.
]<a-z>	Show contents of register.
<count> .	Repeat previous command <count> times. For destructive commands as well as undo.
f1 1 and f1 2	While `.` repeats the last destructive command, these two macros repeat the second-last and the third-last destructive commands. See Section 3.4 [Vi Macros], page 36, for more information on Vi macros.
C-c M-p and C-c M-n	In Vi state, these commands help peruse the history of Vi's destructive commands. Successive typing of *C-c M-p* causes Viper to search the history in the direction of older commands, while hitting *C-c M-n* does so in reverse order. Each command in the history is displayed in the Minibuffer. The displayed command can then be executed by typing '`.`'. Since typing the above sequences of keys may be tedious, the functions doing the perusing can be bound to unused keyboard keys in the '`~/.vip`' file. See Section 3.3 [Viper Specials], page 31, for details.

4.2.4 Editing in Insert State

Minibuffer can be edited similarly to Insert state, and you can switch between Insert/Replace/Vi states at will. Some users prefer plain Emacs feel in the Minibuffer. To this end, set *vip-vi-style-in-minibuffer* to `nil`.

C-v	Deprive the next char of its special meaning (quoting).
C-h	One char back.
C-w	One word back.
C-u	Back to the begin of the change on the current line.

4.2.5 Deleting Text

There is one difference in text deletion that you should be aware of. This difference comes from Emacs and was adopted in Viper because we find it very useful. In Vi, if you delete a line, say, and then another line, these two deletions are separated and are put back separately if you use the ‘`p`’ command. In Emacs (and Viper), successive series of deletions that are *not interrupted* by other commands are lumped together, so the deleted text gets accumulated and can be put back as one chunk. If you want to break a sequence of deletions so that the newly deleted text could be put back separately from the previously deleted text, you should perform a non-deleting action, e.g., move the cursor one character in any direction.

<count> x	Delete <count> chars under and after the cursor.
<count> X	Delete <count> chars before the cursor.
<count> d<move>	Delete from point to endpoint of <count><move>.
<count> dd	Delete <count> lines.
D	The rest of the line.
<count> <<move>	Shift the lines described by <count><move> one shiftwidth to the left (layout!).
<count> <<	Shift <count> lines one shiftwidth to the left.

4.2.6 Changing Text

<count> r<char>	Replace <count> chars by <char> - no <esc>.
<count> R	Overwrite the rest of the line, appending change *count - 1* times.
<count> s	Substitute <count> chars.
<count> S	Change <count> lines.
<count> c<move>	Change from begin to endpoint of <count><move>.
<count> cc	Change <count> lines.
<count> C	The rest of the line and <count> - 1 next lines.
<count> =<move>	Reindent the region described by move.
<count> ~	Switch lower and upper cases.

`<count> J` Join <count> lines (default 2).

`:[x,y]s/<p>/<r>/<f>`

Substitute (on lines x through y) the pattern <p> (default the last pattern) with <r>. Useful flags <f> are ‘`g`’ for ‘`global`’ (i.e. change every non-overlapping occurrence of <p>) and ‘`c`’ for ‘`confirm`’ (type ‘`y`’ to confirm a particular substitution, else ‘`n`’). Instead of / any punctuation CHAR unequal to <space> <tab> and <lf> can be used as delimiter.

`:[x,y]copy [z]`

Copy text between *x* and *y* to the position after *z*.

`:[x,y]t [z]`

Same as `:copy`.

`:[x,y]move [z]`

Move text between *x* and *y* to the position after *z*.

`&` Repeat latest Ex substitute command, e.g. `:s/wrong/good`.

`C-c /` Toggle case-sensitive search. With prefix argument, toggle vanilla/regular expression search.

`#c<move>` Change upper case characters in the region to lower case.

`#C<move>` Change lower case characters in the region to upper case.

`#q<move>` Insert specified string at the beginning of each line in the region

`C-c M-p` and `C-c M-n`

In Insert and Replace states, these keys are bound to commands that peruse the history of the text previously inserted in other insert or replace commands. By repeatedly typing `C-c M-p` or `C-c M-n`, you will cause Viper to insert these previously used strings one by one. When a new string is inserted, the previous one is deleted.

In Vi state, these keys are bound to functions that peruse the history of destructive Vi commands. See Section 3.3 [Viper Specials], page 31, for details.

4.2.7 Search and Replace

See Section 4.1 [Groundwork], page 41, for Ex address syntax. See Section 4.7 [Options], page 54 to see how to get literal (non-regular-expression) search and how to stop search from wrapping around.

`<count> /<string>`

To the <count>th occurrence of <string>.

`<count> ?<string>`

To the <count>th previous occurrence of <string>.

`<count> g<move>`

Search for the text described by move. (off by default)

`n`	Repeat latest `/` `?` (next).
`N`	Idem in opposite direction.
`%`	Find the next bracket and go to its match
`:[x,y]g/<string>/<cmd>`	Search globally [from line x to y] for <string> and execute the Ex <cmd> on each occurrence.
`:[x,y]v/<string>/<cmd>`	Execute <cmd> on the lines that don't match.
`#g<move>`	Execute the last keyboard macro for each line in the region. See Section 2.4 [Macros and Registers], page 15, for more info.
`Q`	Query Replace.
`:ta <name>`	Search in the tags file where <name> is defined (file, line), and go to it.
`:[x,y]s/<p>/<r>/<f>`	Substitute (on lines x through y) the pattern <p> (default the last pattern) with <r>. Useful flags <f> are '`g`' for '`global`' (i.e. change every non-overlapping occurrence of <p>) and '`c`' for '`confirm`' (type '`y`' to confirm a particular substitution, else '`n`'). Instead of `/` any punctuation CHAR unequal to <space> <tab> and <lf> can be used as delimiter.
`&`	Repeat latest Ex substitute command, e.g. `:s/wrong/good`.

4.2.8 Yanking

`<count> y<move>`	Yank from begin to endpoint of <count><move>.
`<count> "<a-z>y<move>`	Yank from begin to endpoint of <count><move> to register.
`<count> "<A-Z>y<move>`	Yank from begin to endpoint of <count><move> and append to register.
`<count> yy`	<count> lines.
`<count> Y`	Idem (should be equivalent to `y$` though).
`m<a-z>`	Mark the cursor position with a letter.
`[<a-z>`	Show contents of textmarker.
`]<a-z>`	Show contents of register.
`<count> ["<a-z1-9>]p`	Put the contents of the (default undo) buffer <count> times after the cursor. The register will be automatically downcased.

`<count> ["<a-z1-9>]P`	Put the contents of the (default undo) buffer <count> times before the cursor. The register will

4.2.9 Undoing

`u U`	Undo the latest change.
`.`	Repeat undo.
`:q!`	Quit Vi without writing.
`:e!`	Re-edit a messed-up file.
`:rec`	Recover file from autosave. Viper also creates backup files that have a ‘~’ appended to them.

4.3 Display

`C-g`	At user level 1, give file name, status, current line number and relative position. At user levels 2 and higher, abort the current command.
`C-c g`	Give file name, status, current line number and relative position – all user levels.
`C-l`	Refresh the screen.
`<count> C-e`	Expose <count> more lines at bottom, cursor stays put (if possible).
`<count> C-y`	Expose <count> more lines at top, cursor stays put (if possible).
`<count> C-d`	Scroll <count> lines downward (default the number of the previous scroll; initialization: half a page).
`<count> C-u`	Scroll <count> lines upward (default the number of the previous scroll; initialization: half a page).
`<count> C-f`	<count> pages forward.
`<count> C-b`	<count> pages backward (in older versions `C-b` only works without count).

`<count> z<cr>`
`zH`
	Put line <count> at the top of the window (default the current line).

`<count> z-`
`zL`
	Put line <count> at the bottom of the window (default the current line).

`<count> z.`
`zM`
	Put line <count> in the center of the window (default the current line).

4.4 File and Buffer Handling

In all file handling commands, space should be typed before entering the file name. If you need to type a modifier, such as `>>` or `!`, don't put any space between the command and the modifier.

`:q`	Quit buffer except if modified.
`:q!`	Quit buffer without checking. In Viper, these two commands are identical. Confirmation is required if exiting modified buffers that visit files.
`:susp`	
`:stop`	Suspend Viper
`:[x,y] w`	Write the file. Viper nakes sure that a final newline is always added to any file where this newline is missing. This is done by setting Emacs variable `require-final-newline` to `t`. If you don't like this feature, use `setq-default` to set `require-final-newline` to `nil`. This must be done either in '`.vip`' file or in `.emacs` after Viper is loaded.
`:[x,y] w <name>`	Write to the file <name>.
`:[x,y] w>> <name>`	Append the buffer to the file <name>. There should be no space between `w` and `>>`. Type space after the `>>` and see what happens.
`:w! <name>`	Overwrite the file <name>. In Viper, `:w` and `:w!` are identical. Confirmation is required for writing to an existing file (if this is not the file the buffer is visiting) or to a read-only file.
`:x,y w <name>`	Write lines x through y to the file <name>.
`:wq`	Write the file and kill buffer.
`:r <file> [<file> ...]`	Read file into a buffer, inserting its contents after the current line.
`:xit`	Same as `:wq`.
`:W`	Save unsaved buffers, asking for confirmation.
`:WW`	Like `W`, but without asking for confirmation.
`ZZ`	Save current buffer and kill it. If user level is 1, then save all files and kill Emacs. Killing Emacs is the wrong way to use it, so you should switch to higher user levels as soon as possible.
`:x [<file>]`	Save and kill buffer.
`:x! [<file>]`	`:w![<file>]` and `:q`.

`:pre`	Preserve the file – autosave buffers.
`:rec`	Recover file from autosave.
`:f`	Print file name and lines.
`:cd [<dir>]`	Set the working directory to <dir> (default home directory).
`:pwd`	Print present working directory.
`:e [+<cmd>] <files>`	Edit files. If no filename is given, edit the file visited by the current buffer. If buffer was modified or the file changed on disk, ask for confirmation. Unlike Vi, Viper allows `:e` to take multiple arguments. The first file is edited the same way as in Vi. The rest are visited in the usual Emacs way.
`:e! [+<cmd>] <files>`	Re-edit file. If no filename, reedit current file. In Viper, unlike Vi, `e!` is identical to `:e`. In both cases, the user is asked to confirm if there is a danger of discarding changes to a buffer.
`:q!`	Quit Vi without writing.
`C-^`	Edit the alternate (normally the previous) file.
`:rew`	Obsolete
`:args`	List files not shown anywhere with counts for next
`:n [count] [+<cmd>] [<files>]`	Edit <count> file, or edit files. The count comes from :args.
`:N [count] [+<cmd>] [<files>]`	Like `:n`, but the meaning of the variable *ex-cycle-other-window* is reversed.
`:b`	Switch to another buffer. If *ex-cycle-other-window* is `t`, switch in another window. Buffer completion is supported.
`:B`	Like `:b`, but the meaning of *ex-cycle-other-window* is reversed.
`:<address>r <name>`	Read the file <name> into the buffer after the line <address>.
`v, V, C-v`	Edit a file in current or another window, or in another frame. File name is typed in Minibuffer. File completion and history are supported.

4.5 Mapping

`:map <string>`	Start defining a Vi-style keyboard macro. For instance, typing `:map www` followed by `:!wc %` and then typing `C-x )` will cause `www` to run wc on current file (Vi replaces '%' with the current file name).

`C-x )`	Finish defining a keyboard macro. In Viper, this command completes the process of defining all keyboard macros, whether they are Emacs-style or Vi-style. This is a departure from Vi, needed to allow WYSIWYG mapping of keyboard macros and to permit the use of function keys and arbitrary Emacs functions in the macros.
`:unmap <string>`	Deprive <string> of its mappings in Vi state.
`:map! <string>`	Map a macro for Insert state.
`:unmap! <string>`	Deprive <string> of its mapping in Insert state (see `:unmap`).
`@<a-z>`	In Vi state, execute the contents of register as a command.
`@@`	In Vi state, repeat last register command.
`@#`	In Vi state, begin keyboard macro. End with `@<a-z>`. This will put the macro in the proper register. Register will be automatically downcased. See Section 2.4 [Macros and Registers], page 15, for more info.
`@!<a-z>`	In Vi state, yank anonymous macro to register
`*`	In Vi state, execute anonymous macro (defined by C-x(and C-x)).
`C-x e`	Like `*`, but works in all Viper states.
`#g<move>`	Execute the last keyboard macro for each line in the region. See Section 2.4 [Macros and Registers], page 15, for more info.
`[<a-z>`	Show contents of textmarker.
`]<a-z>`	Show contents of register.

4.6 Shell Commands

Note that % is used in Ex commands to mean current file. If you want a % in your command, it must be escaped as '`\%`'. However if % is the first character, it stands as the address for the whole file. Similarly, '`#`' expands to the previous file. The previous file is the first file in `:args` listing. This defaults to the previous file in the VI sense if you have one window.

`:sh`	Execute a subshell in another window
`:[x,y]!<cmd>`	Execute a shell <cmd> [on lines x through y; % is replace by current file, \% is changed to %
`:[x,y]!! [<args>]`	Repeat last shell command [and append <args>].
`:!<cmd>`	Just execute command and display result in a buffer.

`:!! <args>`	Repeat last shell command and append <args>
`<count> !<move><cmd>`	The shell executes <cmd>, with standard input the lines described by <count><move>, next the standard output replaces those lines (think of ‘`cb`’, ‘`sort`’, ‘`nroff`’, etc.).
`<count> !!<cmd>`	Give <count> lines as standard input to the shell <cmd>, next let the standard output replace those lines.
`:[x,y] w !<cmd>`	Let lines x to y be standard input for <cmd> (notice the <sp> between `w` and `!`).
`:<address>r !<cmd>`	Put the output of <cmd> after the line <address> (default current).
`:<address>r <name>`	Read the file <name> into the buffer after the line <address> (default current).

4.7 Options

`ai`	autoindent – In append mode after a <cr> the cursor will move directly below the first CHAR on the previous line.
`ic`	ignorecase – No distinction between upper and lower cases when searching.
`magic`	Regular expressions used in searches; nomagic means no regexps.
`ro`	readonly – The file is not to be changed. If the user attempts to write to this file, confirmation will be requested.
`sh=<string>`	shell – The program to be used for shell escapes (default ‘`$SHELL`’ (default ‘`/bin/sh`’)).
`sw=<count>`	shiftwidth – Gives the shiftwidth (default 8 positions).
`sm`	showmatch – Whenever you append a `)`, Vi shows its match if it’s on the same page; also with `{` and `}`. If there’s no match, Vi will beep.
`ts=<count>`	tabstop – The length of a <ht>; warning: this is only IN the editor, outside of it <ht>s have their normal length (default 8 positions).
`wm=<count>`	wrapmargin – In append mode Vi automatically puts a <lf> whenever there is a <sp> or <ht> within <wm> columns from the right margin.

`ws`	wrapscan – When searching, the end is considered '`stuck`' to the begin of the file.
`:set <option>`	Turn <option> on.
`:set no<option>`	Turn <option> off.
`:set <option>=<value>`	Set <option> to <value>.

4.8 Emacs Related Commands

`_`	Begin Meta command in Vi state. Most often used as _x (M-x).
`C-z`	Begin Meta command in Insert state.
`C-z`	Switch between Emacs and Vi states.
`C-x0`	Close Window
`C-x1`	Close Other Windows
`C-x2`	Split Window
`C-xo`	Move among windows
`C-xC-f`	Emacs find-file, useful in Insert state
`C-y`	Put back the last killed text. Similar to Vi's `p`, but also works in Insert and Replace state. This command doesn't work in Vi command state, since this binding is taken for something else.
`M-y`	Undoes the last `C-y` and puts another kill from the kill ring. Using this command, you can try may different kills until you find the one you need.

4.9 Mouse-bound Commands

The following two mouse actions are normally bound to to special search and insert commands in of Viper:

`S-mouse-1 (Emacs)`
`meta button1up (XEmacs)`
	Holding Shift (or Meta, if XEmacs) and clicking mouse button 1 will initiate search for a region under the mouse pointer. This command can take a prefix argument. Note: Viper sets this binding only if this mouse action is not already bound to something else. See Section 3.3 [Viper Specials], page 31, for more information.

`S-mouse-2 (Emacs)`	

`meta button2up (XEmacs)`

Holding Shift (or Meta, if XEmacs) and clicking button 2 of the mouse will insert a region surrounding the mouse pointer. This command can also take a prefix argument. Note: Viper sets this binding only if this mouse action is not already bound to something else. See Section 3.3 [Viper Specials], page 31, for more details.

Acknowledgments

Viper, formerly known as VIP-19, was written by Michael Kifer. Viper is based on the original VIP package by Masahiko Sato and on its enhancement, VIP 4.4, by Aamod Sane. This manual is an adaptation of the manual for VIP 4.4, which, in turn, was based on Sato's manual for VIP 3.5.

Many contributors on the net pointed out bugs and suggested a number of useful features. Here is a (hopefully) complete list of contributors:

> `jjm@hplb.hpl.hp.com (Jean-Jacques Moreau), jl@cse.ogi.edu (John Launchbury), rxga@ulysses.att.com, jamesm@bga.com (D.J. Miller II), ascott@fws214.intel.com (Andy Scott), toma@convex.convex.com, dave@hellgate.utah.edu, cook@biostat.wisc.edu (Tom Cook), lindstro@biostat.wisc.edu (Mary Lindstrom), edmonds@edmonds.home.cs.ubc.ca (Brian Edmonds), mveiga@dit.upm.es (Marcelino Veiga Tuimil), dwight@toolucky.llnl.gov (Dwight Shih), phil_brooks@MENTORG.COM (Phil Brooks), kin@isi.com (Kin Cho), ahg@panix.com (Al Gelders), dwallach@cs.princeton.edu (Dan Wallach), hpz@ibmhpz.aug.ipp-garching.mpg.de (Hans-Peter Zehrfeld), simonb@prl.philips.co.uk (Simon Blanchard), Mark.Bordas@East.Sun.COM (Mark Bordas), gviswana@cs.wisc.edu (Guhan Viswanathan), meyering@comco.com (Jim Meyering), pfister@cs.sunysb.edu (Hanspeter Pfister), amade@diagram.fr (Paul-Bernard Amade), jackr@dblues.engr.sgi.com (Jack Repenning), pogrell@informatik.hu-berlin.de (Lutz Pogrell), csdayton@midway.uchicago.edu (Soren Dayton), pradyut@cs.uchicago.edu (Pradyut Shah), vrenjak@sun1.racal.com (Milan Vrenjak), gvr@halcyon.com (George V. Reilly), whicken@dragon.parasoft.com (Wendell Hicken), terra@diku.dk (Morten Welinder), kanze@gabi-soft.fr (James Kanze), hatazaki@bach.convex.com (Takao Hatazaki), sawdey@lcse.umn.edu (Aaron Sawdey), jobrien@hchp.org (John O'Brien), mrb@Eng.Sun.COM (Martin Buchholz)`

Key Index

Function Index

Variable Index

Package Index

(Index is nonexistent)

Concept Index

Table of Contents