GNU ed

The GNU line editor
for GNU ed version 0.5, 9 March 2007

by Andrew L. Moore and Antonio Diaz Diaz

Copyright © 1993, 2006, 2007 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

Table of Contents

1 Overview 2

2 Introduction to Line Editing 3

3 Invoking Ed 7

4 Line Addressing 8

5 Regular Expressions 10

6 Commands 12

7 Limitations 16

8 Diagnostics 17

Copyright © 1993, 2006, 2007 Free Software Foundation, Inc.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

1 Overview

`ed` is a line-oriented text editor. It is used to create, display, modify and otherwise manipulate text files. `red` is a restricted `ed`: it can only edit files in the current directory and cannot execute shell commands.

If invoked with a *file* argument, then a copy of *file* is read into the editor's buffer. Changes are made to this copy and not directly to *file* itself. Upon quitting `ed`, any changes not explicitly saved with a '`w`' command are lost.

Editing is done in two distinct modes: *command* and *input*. When first invoked, `ed` is in command mode. In this mode commands are read from the standard input and executed to manipulate the contents of the editor buffer. A typical command might look like:

```
,s/old/new/g
```

which replaces all occurences of the string *old* with *new*.

When an input command, such as '`a`' (append), '`i`' (insert) or '`c`' (change), is given, `ed` enters input mode. This is the primary means of adding text to a file. In this mode, no commands are available; instead, the standard input is written directly to the editor buffer. A *line* consists of the text up to and including a ⟨newline⟩ character. Input mode is terminated by entering a single period ('`.`') on a line.

All `ed` commands operate on whole lines or ranges of lines; e.g., the '`d`' command deletes lines; the '`m`' command moves lines, and so on. It is possible to modify only a portion of a line by means of replacement, as in the example above. However even here, the '`s`' command is applied to whole lines at a time.

In general, `ed` commands consist of zero or more line addresses, followed by a single character command and possibly additional parameters; i.e., commands have the structure:

```
[address [,address]]command[parameters]
```

The *address*es indicate the line or range of lines to be affected by the command. If fewer addresses are given than the command accepts, then default addresses are supplied.

2 Introduction to Line Editing

`ed` was created, along with the Unix operating system, by Ken Thompson and Dennis Ritchie. It is the refinement of its more complex, programmable predecessor, *QED*, to which Thompson and Ritchie had already added pattern matching capabilities (see Chapter 5 [Regular Expressions], page 10).

For the purposes of this tutorial, a working knowledge of the Unix shell `sh` (see section "Bash" in *The GNU Bash Reference Manual*) and the Unix file system is recommended, since `ed` is designed to interact closely with them.

The principal difference between line editors and display editors is that display editors provide instant feedback to user commands, whereas line editors require sometimes lengthy input before any effects are seen. The advantage of instant feedback, of course, is that if a mistake is made, it can be corrected immediately, before more damage is done. Editing in `ed` requires more strategy and forethought; but if you are up to the task, it can be quite efficient.

Much of the `ed` command syntax is shared with other Unix utilities.

As with the shell, ⟨RETURN⟩ (the carriage-return key) enters a line of input. So when we speak of "entering" a command or some text in `ed`, ⟨RETURN⟩ is implied at the end of each line. Prior to typing ⟨RETURN⟩, corrections to the line may be made by typing either ⟨BACKSPACE⟩ (sometimes labeled ⟨DELETE⟩ or ⟨DEL⟩) to erase characters backwards, or ⟨CONTROL⟩-u (i.e., hold the CONTROL key and type u) to erase the whole line.

When `ed` first opens, it expects to be told what to do but doesn't prompt us like the shell. So let's begin by telling `ed` to do so with the ⟨P⟩ (*prompt*) command:

```
$ ed
P
*
```

By default, `ed` uses asterisk ('`*`') as command prompt to avoid confusion with the shell command prompt ('`$`').

We can run Unix shell (`sh`) commands from inside `ed` by prefixing them with ⟨!⟩ (exclamation mark, aka "bang"). For example:

```
*!date
Mon Jun 26 10:08:41 PDT 2006
!
*!for s in hello world; do echo $s; done
hello
world
!
*
```

So far, this is no different from running commands in the Unix shell. But let's say we want to edit the output of a command, or save it to a file. First we must capture the command output to a temporary location called a *buffer* where `ed` can access it. This is done with `ed`'s ⟨r⟩ command (mnemonic: *read*):

```
*r !cal
143
```

```
*
```

Here `ed` is telling us that it has just read 139 characters into the editor buffer - i.e., the output of the `cal` command, which prints a simple ASCII calendar. To display the buffer contents we issue the ⟨p⟩ (*print*) command (not to be confused with the prompt command, which is uppercase!). To indicate the range of lines in the buffer that should be printed, we prefix the command with ⟨,⟩ (comma) which is shorthand for "the whole buffer":

```
*,p
   September 2006
Mo Tu We Th Fr Sa Su
              1  2  3
 4  5  6  7  8  9 10
11 12 13 14 15 16 17
18 19 20 21 22 23 24
25 26 27 28 29 30

*
```

Now let's write the buffer contents to a file named `junk` with the ⟨w⟩ (*write*) command. Again, we use the ⟨,⟩ prefix to indicate that it's the whole buffer we want:

```
*,w junk
143
*
```

Need we say? It's good practice to frequently write the buffer contents, since unwritten changes to the buffer will be lost when we exit `ed`.

The sample sessions below illustrate some basic concepts of line editing with `ed`. We begin by creating a file, '`sonnet`', with some help from Shakespeare. As with the shell, all input to `ed` must be followed by a ⟨newline⟩ character. Comments begin with a '`#`'.

```
$ ed
# The 'a' command is for appending text to the editor buffer.
a
No more be grieved at that which thou hast done.
Roses have thorns, and filvers foutians mud.
Clouds and eclipses stain both moon and sun,
And loathsome canker lives in sweetest bud.
.
# Entering a single period on a line returns ed to command mode.
# Now write the buffer to the file 'sonnet' and quit:
w sonnet
183
# ed reports the number of characters written.
q
$ ls -l
total 2
-rw-rw-r--    1 alm           183 Nov 10 01:16 sonnet
$
```

In the next example, some typos are corrected in the file '`sonnet`'.

```
$ ed sonnet
183
# Begin by printing the buffer to the terminal with the ‘p’ command.
# The ‘,’ means ‘‘all lines.’’
,p
No more be grieved at that which thou hast done.
Roses have thorns, and filvers foutians mud.
Clouds and eclipses stain both moon and sun,
And loathsome canker lives in sweetest bud.
# Select line 2 for editing.
2
Roses have thorns, and filvers foutians mud.
# Use the substitute command, ‘s’, to replace ‘filvers’ with ‘silver’,
# and print the result.
s/filvers/silver/p
Roses have thorns, and silver foutians mud.
# And correct the spelling of ‘fountains’.
s/utia/untai/p
Roses have thorns, and silver fountains mud.
w sonnet
183
q
$
```

Since `ed` is line-oriented, we have to tell it which line, or range of lines we want to edit. In the above example, we do this by specifying the line's number, or sequence in the buffer. Alternatively, we could have specified a unique string in the line, e.g., '`/filvers/`', where the '`/`'s delimit the string in question. Subsequent commands affect only the selected line, a.k.a. the *current* line. Portions of that line are then replaced with the substitute command, whose syntax is '`s/`*`old`*`/`*`new`*`/`'.

Although `ed` accepts only one command per line, the print command '`p`' is an exception, and may be appended to the end of most commands.

In the next example, a title is added to our sonnet.

```
$ ed sonnet
183
a
 Sonnet #50
.
,p
No more be grieved at that which thou hast done.
Roses have thorns, and silver fountains mud.
Clouds and eclipses stain both moon and sun,
And loathsome canker lives in sweetest bud.
 Sonnet #50
# The title got appended to the end; we should have used ‘0a’
# to append ‘‘before the first line.’’
# Move the title to its proper place.
```

```
5m0p
 Sonnet #50
# The title is now the first line, and the current line has been
# set to this line as well.
,p
 Sonnet #50
No more be grieved at that which thou hast done.
Roses have thorns, and silver fountains mud.
Clouds and eclipses stain both moon and sun,
And loathsome canker lives in sweetest bud.
wq sonnet
195
$
```

When `ed` opens a file, the current line is initially set to the last line of that file. Similarly, the move command '`m`' sets the current line to the last line moved.

In summary:

Structurally, Related programs or routines are `vi (1)`, `sed (1)`, `regex (3)`, `sh (1)`. Relevant documents are:

Unix User's Manual Supplementary Documents: 12 — 13

B. W. Kernighan and P. J. Plauger: "Software Tools in Pascal", Addison-Wesley, 1981.

3 Invoking Ed

The format for running `ed` is:

```
ed [options] [file]
red [options] [file]
```

file specifies the name of a file to read. If *file* is prefixed with a bang (!), then it is interpreted as a shell command. In this case, what is read is the standard output of *file* executed via `sh (1)`. To read a file whose name begins with a bang, prefix the name with a backslash (\). The default filename is set to *file* only if it is not prefixed with a bang.

`ed` supports the following options:

`--help`
`-h`
: Print an informative help message describing the options and exit.

`--version`
`-V`
: Print the version number of `ed` on the standard output and exit.

`--loose-exit-status`
`-l`
: Do not exit with bad status if a command happens to "fail" (for example if a substitution command finds nothing to replace). This can be useful when `ed` is invoked as the editor for crontab.

`--prompt=`*string*
`-p` *string*
: Specifies a command prompt. This may be toggled on and off with the '`P`' command.

`--quiet`
`--silent`
`-s`
: Suppresses diagnostics. This should be used if `ed`'s standard input is from a script.

`--traditional`
`-G`
: Forces backwards compatibility. This affects the behavior of the `ed` commands '`G`', '`V`', '`f`', '`l`', '`m`', '`t`' and '`!!`'. If the default behavior of these commands does not seem familiar, then try invoking `ed` with this switch.

`--verbose`
`-v`
: Verbose mode. This may be toggled on and off with the '`H`' command.

4 Line Addressing

An address represents the number of a line in the buffer. `ed` maintains a *current address* which is typically supplied to commands as the default address when none is specified. When a file is first read, the current address is set to the last line of the file. In general, the current address is set to the last line affected by a command.

A line address is constructed from one of the bases in the list below, optionally followed by a numeric offset. The offset may include any combination of digits, operators (i.e., '`+`' and '`-`') and whitespace. Addresses are read from left to right, and their values may be absolute or relative to the current address.

One exception to the rule that addresses represent line numbers is the address '`0`' (zero). This means "before the first line," and is valid wherever it makes sense.

An address range is two addresses separated either by a comma or semicolon. The value of the first address in a range cannot exceed the value of the the second. If only one address is given in a range, then the second address is set to the given address. If an *n*-tuple of addresses is given where $n > 2$, then the corresponding range is determined by the last two addresses in the *n*-tuple. If only one address is expected, then the last address is used.

Each address in a comma-delimited range is interpreted relative to the current address. In a semicolon-delimited range, the first address is used to set the current address, and the second address is interpreted relative to the first.

The following address symbols are recognized.

`.` The current line (address) in the buffer.

`$` The last line in the buffer.

`n` The *n*th, line in the buffer where *n* is a number in the range '`0,$`'.

`+` The next line. This is equivalent to '`+1`' and may be repeated with cumulative effect.

`-` The previous line. This is equivalent to '`-1`' and may be repeated with cumulative effect.

`+n`
`whitespace n`
The *n*th next line, where *n* is a non-negative number. Whitespace followed by a number *n* is interpreted as '*`+n`*'.

`-n` The *n*th previous line, where *n* is a non-negative number.

`,` The first through last lines in the buffer. This is equivalent to the address range '`1,$`'.

`;` The current through last lines in the buffer. This is equivalent to the address range '`.,$`'.

`/re/` The next line containing the regular expression *re*. The search wraps to the beginning of the buffer and continues down to the current line, if necessary. '`//`' repeats the last search.

`?re?` The previous line containing the regular expression *re*. The search wraps to the end of the buffer and continues up to the current line, if necessary. '`??`' repeats the last search.

`'x` The apostrophe-x character pair addresses the line previously marked by a '`k`' (mark) command, where '`x`' is a lower case letter from the portable character set.

5 Regular Expressions

Regular expressions are patterns used in selecting text. For example, the `ed` command

```
g/string/
```

prints all lines containing *string*. Regular expressions are also used by the '`s`' command for selecting old text to be replaced with new text.

In addition to a specifying string literals, regular expressions can represent classes of strings. Strings thus represented are said to be matched by the corresponding regular expression. If it is possible for a regular expression to match several strings in a line, then the left-most longest match is the one selected.

The following symbols are used in constructing regular expressions:

c Any character *c* not listed below, including '`{`', '`}`', '`(`', '`)`', '`<`' and '`>`', matches itself.

`\`*c* Any backslash-escaped character *c*, other than '`{`', "`}`", '`(`', '`)`', '`<`', '`>`', '`b`', '`B`', '`w`', '`W`', '`+`' and '`?`', matches itself.

`.` Matches any single character.

`[`*char-class*`]`
Matches any single character in *char-class*. To include a '`]`' in *char-class*, it must be the first character. A range of characters may be specified by separating the end characters of the range with a '`-`', e.g., '`a-z`' specifies the lower case characters. The following literal expressions can also be used in *char-class* to specify sets of characters:

```
[:alnum:] [:cntrl:] [:lower:] [:space:]
[:alpha:] [:digit:] [:print:] [:upper:]
[:blank:] [:graph:] [:punct:] [:xdigit:]
```

If '`-`' appears as the first or last character of *char-class*, then it matches itself. All other characters in *char-class* match themselves.

Patterns in *char-class* of the form:

```
[.col-elm.]
[=col-elm=]
```

where *col-elm* is a *collating element* are interpreted according to `locale (5)`. See `regex (3)` for an explanation of these constructs.

`[^`*char-class*`]`
Matches any single character, other than newline, not in *char-class*. *char-class* is defined as above.

`^` If '`^`' is the first character of a regular expression, then it anchors the regular expression to the beginning of a line. Otherwise, it matches itself.

`$` If '`$`' is the last character of a regular expression, it anchors the regular expression to the end of a line. Otherwise, it matches itself.

`\(`*re*`\)` Defines a (possibly null) subexpression *re*. Subexpressions may be nested. A subsequent backreference of the form '`\`*n*', where *n* is a number in the range

[1,9], expands to the text matched by the *n*th subexpression. For example, the regular expression '`\(a.c\)\1`' matches the string '`abcabc`', but not '`abcadc`'. Subexpressions are ordered relative to their left delimiter.

`*` Matches the single character regular expression or subexpression immediately preceding it zero or more times. If '`*`' is the first character of a regular expression or subexpression, then it matches itself. The '`*`' operator sometimes yields unexpected results. For example, the regular expression '`b*`' matches the beginning of the string '`abbb`', as opposed to the substring '`bbb`', since a null match is the only left-most match.

`\{`*n*`,`*m*`\}`
`\{`*n*`,\}`
`\{`*n*`\}` Matches the single character regular expression or subexpression immediately preceding it at least *n* and at most *m* times. If *m* is omitted, then it matches at least *n* times. If the comma is also omitted, then it matches exactly *n* times. If any of these forms occurs first in a regular expression or subexpression, then it is interpreted literally (i.e., the regular expression '`\{2\}`' matches the string '`{2}`', and so on).

`\<`
`\>` Anchors the single character regular expression or subexpression immediately following it to the beginning (in the case of '`\<`') or ending (in the case of '`\>`') of a *word*, i.e., in ASCII, a maximal string of alphanumeric characters, including the underscore (_).

The following extended operators are preceded by a backslash '`\`' to distinguish them from traditional `ed` syntax.

`\‘`
`\’` Unconditionally matches the beginning '`\‘`' or ending '`\’`' of a line.

`\?` Optionally matches the single character regular expression or subexpression immediately preceding it. For example, the regular expression '`a[bd]\?c`' matches the strings '`abc`', '`adc`' and '`ac`'. If '`\?`' occurs at the beginning of a regular expressions or subexpression, then it matches a literal '`?`'.

`\+` Matches the single character regular expression or subexpression immediately preceding it one or more times. So the regular expression '`a+`' is shorthand for '`aa*`'. If '`\+`' occurs at the beginning of a regular expression or subexpression, then it matches a literal '`+`'.

`\b` Matches the beginning or ending (null string) of a word. Thus the regular expression '`\bhello\b`' is equivalent to '`\<hello\>`'. However, '`\b\b`' is a valid regular expression whereas '`\<\>`' is not.

`\B` Matches (a null string) inside a word.

`\w` Matches any character in a word.

`\W` Matches any character not in a word.

6 Commands

All `ed` commands are single characters, though some require additonal parameters. If a command's parameters extend over several lines, then each line except for the last must be terminated with a backslash ('\').

In general, at most one command is allowed per line. However, most commands accept a print suffix, which is any of '`p`' (print), '`l`' (list), or '`n`' (enumerate), to print the last line affected by the command.

An interrupt (typically ⟨Control-C⟩) has the effect of aborting the current command and returning the editor to command mode.

`ed` recognizes the following commands. The commands are shown together with the default address or address range supplied if none is specified (in parenthesis).

`(.)a`
: Appends text to the buffer after the addressed line, which may be the address '`0`' (zero). Text is entered in input mode. The current address is set to last line entered.

`(.,.)c`
: Changes lines in the buffer. The addressed lines are deleted from the buffer, and text is appended in their place. Text is entered in input mode. The current address is set to last line entered.

`(.,.)d`
: Deletes the addressed lines from the buffer. If there is a line after the deleted range, then the current address is set to this line. Otherwise the current address is set to the line before the deleted range.

`e` *file*
: Edits *file*, and sets the default filename. If *file* is not specified, then the default filename is used. Any lines in the buffer are deleted before the new file is read. The current address is set to the last line read.

`e !`*command*
: Edits the standard output of '`!`*command*', (see the '`!`' command below). The default filename is unchanged. Any lines in the buffer are deleted before the output of *command* is read. The current address is set to the last line read.

`E` *file*
: Edits *file* unconditionally. This is similar to the '`e`' command, except that unwritten changes are discarded without warning. The current address is set to the last line read.

`f` *file*
: Sets the default filename to *file*. If *file* is not specified, then the default unescaped filename is printed.

`(1,$)g /`*re*`/`*command-list*
: Global command. Applies *command-list* to each of the addressed lines matching a regular expression *re*. The current address is set to the line currently matched before *command-list* is executed. At the end of the '`g`' command, the current address is set to the last line affected by *command-list*.

  At least the first command of *command-list* must appear on the same line as the '`g`' command. All lines of a multi-line *command-list* except the last line must be terminated with a backslash ('\'). Any commands are allowed, except for '`g`', '`G`', '`v`', and '`V`'. By default, a newline alone in *command-list* is equivalent

to a ‘`p`’ command. If `ed` is invoked with the command-line option ‘`-G`’, then a newline in *command-list* is equivalent to a ‘`.+1p`’ command.

`(1,$)G /`*re*`/`
: Interactive global command. Interactively edits the addressed lines matching a regular expression *re*. For each matching line, the line is printed, the current address is set, and the user is prompted to enter a *command-list*. At the end of the ‘`G`’ command, the current address is set to the last line affected by (the last) *command-list*.

  The format of *command-list* is the same as that of the ‘`g`’ command. A newline alone acts as a null command list. A single ‘`&`’ repeats the last non-null command list.

`H`
: Toggles the printing of error explanations. By default, explanations are not printed. It is recommended that ed scripts begin with this command to aid in debugging.

`h`
: Prints an explanation of the last error.

`(.)i`
: Inserts text in the buffer before the current line. The address ‘`0`’ (zero) is valid for this command; it is equivalent to address ‘`1`’. Text is entered in input mode. The current address is set to the last line entered.

`(.,.+1)j`
: Joins the addressed lines. The addressed lines are deleted from the buffer and replaced by a single line containing their joined text. The current address is set to the resultant line.

`(.)kx`
: Marks a line with a lower case letter ‘`x`’. The line can then be addressed as ‘`'x`’ (i.e., a single quote followed by ‘`x`’) in subsequent commands. The mark is not cleared until the line is deleted or otherwise modified.

`(.,.)l`
: Prints the addressed lines unambiguously. The end of each line is marked with a ‘`$`’, and every ‘`$`’ character within the text is printed with a preceding backslash. The current address is set to the last line printed.

`(.,.)m(.)`
: Moves lines in the buffer. The addressed lines are moved to after the right-hand destination address, which may be the address ‘`0`’ (zero). The current address is set to the last line moved.

`(.,.)n`
: Prints the addressed lines, preceding each line by its line number and a ⟨tab⟩. The current address is set to the last line printed.

`(.,.)p`
: Prints the addressed lines. The current address is set to the last line printed.

`P`
: Toggles the command prompt on and off. Unless a prompt is specified with command-line option ‘`-p`’, the command prompt is by default turned off.

`q`
: Quits `ed`.

`Q`
: Quits `ed` unconditionally. This is similar to the `q` command, except that unwritten changes are discarded without warning.

`($)r` *file*

Reads *file* to after the addressed line. If *file* is not specified, then the default filename is used. If there is no default filename prior to the command, then the default filename is set to *file*. Otherwise, the default filename is unchanged. The current address is set to the last line read.

`($)r !`*command*

Reads to after the addressed line the standard output of '`!command`', (see the '`!`' command below). The default filename is unchanged. The current address is set to the last line read.

`(.,.)s /`*re*`/`*replacement*`/`
`(.,.)s /`*re*`/`*replacement*`/g`
`(.,.)s /`*re*`/`*replacement*`/n`

Replaces text in the addressed lines matching a regular expression *re* with *replacement*. By default, only the first match in each line is replaced. If the '`g`' (global) suffix is given, then every match is replaced. The *n* suffix, where *n* is a postive number, causes only the *n*th match to be replaced. It is an error if no substitutions are performed on any of the addressed lines. The current address is set the last line affected.

re and *replacement* may be delimited by any character other than ⟨space⟩ and ⟨newline⟩ (see the '`s`' command below). If one or two of the last delimiters is omitted, then the last line affected is printed as if the print suffix '`p`' were specified.

An unescaped '`&`' in *replacement* is replaced by the currently matched text. The character sequence '`\`*m*' where *m* is a number in the range [1,9], is replaced by the *m*th backreference expression of the matched text. If *replacement* consists of a single '`%`', then *replacement* from the last substitution is used. Newlines may be embedded in *replacement* if they are escaped with a backslash ('\').

`(.,.)s` Repeats the last substitution. This form of the '`s`' command accepts a count suffix *n*, or any combination of the characters '`r`', '`g`', and '`p`'. If a count suffix *n* is given, then only the *n*th match is replaced. The '`r`' suffix causes the regular expression of the last search to be used instead of the that of the last substitution. The '`g`' suffix toggles the global suffix of the last substitution. The '`p`' suffix toggles the print suffix of the last substitution The current address is set to the last line affected.

`(.,.)t(.)`

Copies (i.e., transfers) the addressed lines to after the right-hand destination address, which may be the address '`0`' (zero). The current address is set to the last line copied.

`u` Undoes the last command and restores the current address to what it was before the command. The global commands '`g`', '`G`', '`v`', and '`V`' are treated as a single command by undo. '`u`' is its own inverse.

`(1,$)v /`*re*`/`*command-list*

This is similar to the '`g`' command except that it applies *command-list* to each of the addressed lines not matching the regular expression *re*.

`(1,$)V /re/`
: This is similar to the '`G`' command except that it interactively edits the addressed lines not matching the regular expression *re*.

`(1,$)w file`
: Writes the addressed lines to *file*. Any previous contents of *file* is lost without warning. If there is no default filename, then the default filename is set to *file*, otherwise it is unchanged. If no filename is specified, then the default filename is used. The current address is unchanged.

`(1,$)w !command`
: Writes the addressed lines to the standard input of '`!command`', (see the '`!`' command below). The default filename and current address are unchanged.

`(1,$)wq file`
: Writes the addressed lines to *file*, and then executes a '`q`' command.

`(1,$)W file`
: Appends the addressed lines to the end of *file*. This is similar to the '`w`' command, expect that the previous contents of file is not clobbered. The current address is unchanged.

`(.)x` Copies (puts) the contents of the cut buffer to after the addressed line. The current address is set to the last line copied.

`(.,.)y` Copies (yanks) the addressed lines to the cut buffer. The cut buffer is overwritten by subsequent '`y`', '`s`', '`j`', '`d`', or '`c`' commands. The current address is unchanged.

`(.+1)z n` Scrolls *n* lines at a time starting at addressed line. If *n* is not specified, then the current window size is used. The current address is set to the last line printed.

`!command` Executes *command* via `sh (1)`. If the first character of *command* is '`!`', then it is replaced by text of the previous '`!command`'. `ed` does not process *command* for backslash ('\') escapes. However, an unescaped '`%`' is replaced by the default filename. When the shell returns from execution, a '`!`' is printed to the standard output. The current line is unchanged.

`(.,.)#` Begins a comment; the rest of the line, up to a newline, is ignored. If a line address followed by a semicolon is given, then the current address is set to that address. Otherwise, the current address is unchanged.

`($)=` Prints the line number of the addressed line.

`(.+1)`⟨newline⟩
: An address alone prints the addressed line. A ⟨newline⟩ alone is equivalent to '`+1p`'. the current address is set to the address of the printed line.

7 Limitations

If the terminal hangs up, `ed` attempts to write the buffer to file '`ed.hup`'.

`ed` processes *file* arguments for backslash escapes, i.e., in a filename, any characters preceded by a backslash ('\') are interpreted literally.

If a text (non-binary) file is not terminated by a newline character, then `ed` appends one on reading/writing it. In the case of a binary file, `ed` does not append a newline on reading/writing.

Per line overhead: 4 `int`s.

8 Diagnostics

When an error occurs, if `ed`'s input is from a regular file or here document, then it exits, otherwise it prints a '`?`' and returns to command mode. An explanation of the last error can be printed with the '`h`' (help) command.

If the '`u`' (undo) command occurs in a global command list, then the command list is executed only once.

Attempting to quit `ed` or edit another file before writing a modified buffer results in an error. If the command is entered a second time, it succeeds, but any changes to the buffer are lost.

`ed` exits with 0 if no errors occurred; otherwise >0.