2 Invoking grep

The general synopsis of the `grep` command line is

```
grep [option...] [patterns] [file...]
```

There can be zero or more *option* arguments, and zero or more *file* arguments. The *patterns* argument contains one or more patterns separated by newlines, and is omitted when patterns are given via the '`-e patterns`' or '`-f file`' options. Typically *patterns* should be quoted when `grep` is used in a shell command.

2.1 Command-line Options

`grep` comes with a rich set of options: some from POSIX and some being GNU extensions. Long option names are always a GNU extension, even for options that are from POSIX specifications. Options that are specified by POSIX, under their short names, are explicitly marked as such to facilitate POSIX-portable programming. A few option names are provided for compatibility with older or more exotic implementations.

Several additional options control which variant of the `grep` matching engine is used. See Section 2.4 [grep Programs], page 12.

2.1.1 Generic Program Information

`--help` Print a usage message briefly summarizing the command-line options and the bug-reporting address, then exit.

`-V`
`--version`
Print the version number of `grep` to the standard output stream. This version number should be included in all bug reports.

2.1.2 Matching Control

`-e` *patterns*
`--regexp=`*patterns*
Use *patterns* as one or more patterns; newlines within *patterns* separate each pattern from the next. If this option is used multiple times or is combined with the `-f` (`--file`) option, search for all patterns given. Typically *patterns* should be quoted when `grep` is used in a shell command. (`-e` is specified by POSIX.)

`-f` *file*
`--file=`*file*
Obtain patterns from *file*, one per line. If this option is used multiple times or is combined with the `-e` (`--regexp`) option, search for all patterns given. When *file* is '`-`', read patterns from standard input. The empty file contains zero patterns, and therefore matches nothing. (`-f` is specified by POSIX.)

`-i`
`-y`
`--ignore-case`
Ignore case distinctions in patterns and input data, so that characters that differ only in case match each other. Although this is straightforward when letters

differ in case only via lowercase-uppercase pairs, the behavior is unspecified in other situations. For example, uppercase “S” has an unusual lowercase counterpart “ſ” (Unicode character U+017F, LATIN SMALL LETTER LONG S) in many locales, and it is unspecified whether this unusual character matches “S” or “s” even though uppercasing it yields “S”. Another example: the lowercase German letter “ß” (U+00DF, LATIN SMALL LETTER SHARP S) is normally capitalized as the two-character string “SS” but it does not match “SS”, and it might not match the uppercase letter (U+1E9E, LATIN CAPITAL LETTER SHARP S) even though lowercasing the latter yields the former.

`-y` is an obsolete synonym that is provided for compatibility. (`-i` is specified by POSIX.)

`--no-ignore-case`
: Do not ignore case distinctions in patterns and input data. This is the default. This option is useful for passing to shell scripts that already use `-i`, in order to cancel its effects because the two options override each other.

`-v`
`--invert-match`
: Invert the sense of matching, to select non-matching lines. (`-v` is specified by POSIX.)

`-w`
`--word-regexp`
: Select only those lines containing matches that form whole words. The test is that the matching substring must either be at the beginning of the line, or preceded by a non-word constituent character. Similarly, it must be either at the end of the line or followed by a non-word constituent character. Word constituent characters are letters, digits, and the underscore. This option has no effect if `-x` is also specified.

  Because the `-w` option can match a substring that does not begin and end with word constituents, it differs from surrounding a regular expression with ‘`\<`’ and ‘`\>`’. For example, although ‘`grep -w @`’ matches a line containing only ‘`@`’, ‘`grep '\<@\>'`’ cannot match any line because ‘`@`’ is not a word constituent. See Section 3.3 [Special Backslash Expressions], page 16.

`-x`
`--line-regexp`
: Select only those matches that exactly match the whole line. For regular expression patterns, this is like parenthesizing each pattern and then surrounding it with ‘`^`’ and ‘`$`’. (`-x` is specified by POSIX.)

2.1.3 General Output Control

`-c`
`--count` Suppress normal output; instead print a count of matching lines for each input file. With the `-v` (`--invert-match`) option, count non-matching lines. (`-c` is specified by POSIX.)

`--color[=WHEN]`
`--colour[=WHEN]`
: Surround matched non-empty strings, matching lines, context lines, file names, line numbers, byte offsets, and separators (for fields and groups of context lines) with escape sequences to display them in color on the terminal. The colors are defined by the environment variable `GREP_COLORS` and default to '`ms=01;31:mc=01;31:sl=:cx=:fn=35:ln=32:bn=32:se=36`' for bold red matched text, magenta file names, green line numbers, green byte offsets, cyan separators, and default terminal colors otherwise. See Section 2.2 [Environment Variables], page 9.

  WHEN is '`always`' to use colors, '`never`' to not use colors, or '`auto`' to use colors if standard output is associated with a terminal device and the `TERM` environment variable's value suggests that the terminal supports colors. Plain `--color` is treated like `--color=auto`; if no `--color` option is given, the default is `--color=never`.

`-L`
`--files-without-match`
: Suppress normal output; instead print the name of each input file from which no output would normally have been printed.

`-l`
`--files-with-matches`
: Suppress normal output; instead print the name of each input file from which output would normally have been printed. Scanning each input file stops upon first match. (`-l` is specified by POSIX.)

`-m` *num*
`--max-count=`*num*
: Stop after the first *num* selected lines. If *num* is zero, `grep` stops right away without reading input. A *num* of −1 is treated as infinity and `grep` does not stop; this is the default.

  If the input is standard input from a regular file, and *num* selected lines are output, `grep` ensures that the standard input is positioned just after the last selected line before exiting, regardless of the presence of trailing context lines. This enables a calling process to resume a search. For example, the following shell script makes use of it:

```
while grep -m 1 'PATTERN'
do
  echo xxxx
done < FILE
```

  But the following probably will not work because a pipe is not a regular file:

```
# This probably will not work.
cat FILE |
while grep -m 1 'PATTERN'
do
  echo xxxx
done
```

When grep stops after *num* selected lines, it outputs any trailing context lines. When the -c or --count option is also used, grep does not output a count greater than *num*. When the -v or --invert-match option is also used, grep stops after outputting *num* non-matching lines.

-o
--only-matching
: Print only the matched non-empty parts of matching lines, with each such part on a separate output line. Output lines use the same delimiters as input, and delimiters are null bytes if -z (--null-data) is also used (see Section 2.1.7 [Other Options], page 9).

-q
--quiet
--silent
: Quiet; do not write anything to standard output. Exit immediately with zero status if any match is found, even if an error was detected. Also see the -s or --no-messages option. Portability note: Solaris 10 grep lacks -q; portable shell scripts typically can redirect standard output to /dev/null instead of using -q. (-q is specified by POSIX.)

-s
--no-messages
: Suppress error messages about nonexistent or unreadable files. (-s is specified by POSIX.)

2.1.4 Output Line Prefix Control

When several prefix fields are to be output, the order is always file name, line number, and byte offset, regardless of the order in which these options were specified.

-b
--byte-offset
: Print the 0-based byte offset within the input file before each line of output. If -o (--only-matching) is specified, print the offset of the matching part itself.

-H
--with-filename
: Print the file name for each match. This is the default when there is more than one file to search.

-h
--no-filename
: Suppress the prefixing of file names on output. This is the default when there is only one file (or only standard input) to search.

--label=*LABEL*
: Display input actually coming from standard input as input coming from file *LABEL*. This can be useful for commands that transform a file's contents before searching; e.g.:

```
gzip -cd foo.gz | grep --label=foo -H 'some pattern'
```

`-n`
`--line-number`
: Prefix each line of output with the 1-based line number within its input file. (`-n` is specified by POSIX.)

`-T`
`--initial-tab`
: Make sure that the first character of actual line content lies on a tab stop, so that the alignment of tabs looks normal. This is useful with options that prefix their output to the actual content: `-H`, `-n`, and `-b`. This may also prepend spaces to output line numbers and byte offsets so that lines from a single file all start at the same column.

`-Z`
`--null`
: Output a zero byte (the ASCII NUL character) instead of the character that normally follows a file name. For example, '`grep -lZ`' outputs a zero byte after each file name instead of the usual newline. This option makes the output unambiguous, even in the presence of file names containing unusual characters like newlines. This option can be used with commands like '`find -print0`', '`perl -0`', '`sort -z`', and '`xargs -0`' to process arbitrary file names, even those that contain newline characters.

2.1.5 Context Line Control

Context lines are non-matching lines that are near a matching line. They are output only if one of the following options are used. Regardless of how these options are set, `grep` never outputs any given line more than once. If the `-o` (`--only-matching`) option is specified, these options have no effect and a warning is given upon their use.

`-A` *num*
`--after-context=`*num*
: Print *num* lines of trailing context after matching lines.

`-B` *num*
`--before-context=`*num*
: Print *num* lines of leading context before matching lines.

`-C` *num*
`-`*num*
`--context=`*num*
: Print *num* lines of leading and trailing output context.

`--group-separator=`*string*
: When `-A`, `-B` or `-C` are in use, print *string* instead of `--` between groups of lines.

`--no-group-separator`
: When `-A`, `-B` or `-C` are in use, do not print a separator between groups of lines.

Here are some points about how `grep` chooses the separator to print between prefix fields and line content:

- Matching lines normally use '`:`' as a separator between prefix fields and actual line content.

- Context (i.e., non-matching) lines use '-' instead.
- When context is not specified, matching lines are simply output one right after another.
- When context is specified, lines that are adjacent in the input form a group and are output one right after another, while by default a separator appears between non-adjacent groups.
- The default separator is a '--' line; its presence and appearance can be changed with the options above.
- Each group may contain several matching lines when they are close enough to each other that two adjacent groups connect and can merge into a single contiguous one.

2.1.6 File and Directory Selection

`-a`
`--text` Process a binary file as if it were text; this is equivalent to the '`--binary-files=text`' option.

`--binary-files=`*type*

If a file's data or metadata indicate that the file contains binary data, assume that the file is of type *type*. Non-text bytes indicate binary data; these are either output bytes that are improperly encoded for the current locale (see Section 2.2 [Environment Variables], page 9), or null input bytes when the `-z` (`--null-data`) option is not given (see Section 2.1.7 [Other Options], page 9).

By default, *type* is '`binary`', and `grep` suppresses output after null input binary data is discovered, and suppresses output lines that contain improperly encoded data. When some output is suppressed, `grep` follows any output with a message to standard error saying that a binary file matches.

If *type* is '`without-match`', when `grep` discovers null input binary data it assumes that the rest of the file does not match; this is equivalent to the `-I` option.

If *type* is '`text`', `grep` processes binary data as if it were text; this is equivalent to the `-a` option.

When *type* is '`binary`', `grep` may treat non-text bytes as line terminators even without the `-z` (`--null-data`) option. This means choosing '`binary`' versus '`text`' can affect whether a pattern matches a file. For example, when *type* is '`binary`' the pattern '`q$`' might match '`q`' immediately followed by a null byte, even though this is not matched when *type* is '`text`'. Conversely, when *type* is '`binary`' the pattern '`.`' (period) might not match a null byte.

Warning: The `-a` (`--binary-files=text`) option might output binary garbage, which can have nasty side effects if the output is a terminal and if the terminal driver interprets some of it as commands. On the other hand, when reading files whose text encodings are unknown, it can be helpful to use `-a` or to set '`LC_ALL='C''` in the environment, in order to find more matches even if the matches are unsafe for direct display.

`-D` *action*
`--devices=`*action*
: If an input file is a device, FIFO, or socket, use *action* to process it. If *action* is '`read`', all devices are read just as if they were ordinary files. If *action* is '`skip`', devices, FIFOs, and sockets are silently skipped. By default, devices are read if they are on the command line or if the `-R` (`--dereference-recursive`) option is used, and are skipped if they are encountered recursively and the `-r` (`--recursive`) option is used. This option has no effect on a file that is read via standard input.

`-d` *action*
`--directories=`*action*
: If an input file is a directory, use *action* to process it. By default, *action* is '`read`', which means that directories are read just as if they were ordinary files (some operating systems and file systems disallow this, and will cause `grep` to print error messages for every directory or silently skip them). If *action* is '`skip`', directories are silently skipped. If *action* is '`recurse`', `grep` reads all files under each directory, recursively, following command-line symbolic links and skipping other symlinks; this is equivalent to the `-r` option.

`--exclude=`*glob*
: Skip any command-line file with a name suffix that matches the pattern *glob*, using wildcard matching; a name suffix is either the whole name, or a trailing part that starts with a non-slash character immediately after a slash ('`/`') in the name. When searching recursively, skip any subfile whose base name matches *glob*; the base name is the part after the last slash. A pattern can use '`*`', '`?`', and '`[`...'`]`' as wildcards, and `\` to quote a wildcard or backslash character literally.

`--exclude-from=`*file*
: Skip files whose name matches any of the patterns read from *file* (using wildcard matching as described under `--exclude`).

`--exclude-dir=`*glob*
: Skip any command-line directory with a name suffix that matches the pattern *glob*. When searching recursively, skip any subdirectory whose base name matches *glob*. Ignore any redundant trailing slashes in *glob*.

`-I`
: Process a binary file as if it did not contain matching data; this is equivalent to the '`--binary-files=without-match`' option.

`--include=`*glob*
: Search only files whose name matches *glob*, using wildcard matching as described under `--exclude`. If contradictory `--include` and `--exclude` options are given, the last matching one wins. If no `--include` or `--exclude` options match, a file is included unless the first such option is `--include`.

`-r`
`--recursive`
: For each directory operand, read and process all files in that directory, recursively. Follow symbolic links on the command line, but skip symlinks that are

encountered recursively. Note that if no file operand is given, grep searches the working directory. This is the same as the '`--directories=recurse`' option.

`-R`
`--dereference-recursive`
For each directory operand, read and process all files in that directory, recursively, following all symbolic links.

2.1.7 Other Options

`--` Delimit the option list. Later arguments, if any, are treated as operands even if they begin with '`-`'. For example, '`grep PAT -- -file1 file2`' searches for the pattern PAT in the files named `-file1` and `file2`.

`--line-buffered`
Use line buffering for standard output, regardless of output device. By default, standard output is line buffered for interactive devices, and is fully buffered otherwise. With full buffering, the output buffer is flushed when full; with line buffering, the buffer is also flushed after every output line. The buffer size is system dependent.

`-U`
`--binary` On platforms that distinguish between text and binary I/O, use the latter when reading and writing files other than the user's terminal, so that all input bytes are read and written as-is. This overrides the default behavior where `grep` follows the operating system's advice whether to use text or binary I/O. On MS-Windows when `grep` uses text I/O it reads a carriage return–newline pair as a newline and a Control-Z as end-of-file, and it writes a newline as a carriage return–newline pair.

When using text I/O `--byte-offset` (`-b`) counts and `--binary-files` heuristics apply to input data after text-I/O processing. Also, the `--binary-files` heuristics need not agree with the `--binary` option; that is, they may treat the data as text even if `--binary` is given, or vice versa. See Section 2.1.6 [File and Directory Selection], page 7.

This option has no effect on GNU and other POSIX-compatible platforms, which do not distinguish text from binary I/O.

`-z`
`--null-data`
Treat input and output data as sequences of lines, each terminated by a zero byte (the ASCII NUL character) instead of a newline. Like the `-Z` or `--null` option, this option can be used with commands like '`sort -z`' to process arbitrary file names.

2.2 Environment Variables

The behavior of `grep` is affected by several environment variables, the most important of which control the locale, which specifies how `grep` interprets characters in its patterns and data.

The locale for category LC_*foo* is specified by examining the three environment variables LC_ALL, LC_*foo*, and LANG, in that order. The first of these variables that is set specifies the locale. For example, if LC_ALL is not set, but LC_COLLATE is set to 'pt_BR.UTF-8', then a Brazilian Portuguese locale is used for the LC_COLLATE category. As a special case for LC_MESSAGES only, the environment variable LANGUAGE can contain a colon-separated list of languages that overrides the three environment variables that ordinarily specify the LC_MESSAGES category. The 'C' locale is used if none of these environment variables are set, if the locale catalog is not installed, or if grep was not compiled with national language support (NLS). The shell command locale -a lists locales that are currently available.

The following environment variables affect the behavior of grep.

GREP_COLOR

This obsolescent variable interacts with GREP_COLORS confusingly, and grep warns if it is set and is not overridden by GREP_COLORS. Instead of 'GREP_COLOR='*color*'', you can use 'GREP_COLORS='mt=*color*''.

GREP_COLORS

This variable controls how the --color option highlights output. Its value is a colon-separated list of terminfo capabilities that defaults to 'ms=01;31:mc=01;31:sl=:cx=:fn=35:ln=32:bn=32:se=36' with the 'rv' and 'ne' boolean capabilities omitted (i.e., false). The two-letter capability names refer to terminal "capabilities," the ability of a terminal to highlight text, or change its color, and so on. These capabilities are stored in an online database and accessed by the terminfo library. Non-empty capability values control highlighting using Select Graphic Rendition (SGR) commands interpreted by the terminal or terminal emulator. (See the section in the documentation of your text terminal for permitted values and their meanings as character attributes.) These substring values are integers in decimal representation and can be concatenated with semicolons. grep takes care of assembling the result into a complete SGR sequence ('\33['...'m'). Common values to concatenate include '1' for bold, '4' for underline, '5' for blink, '7' for inverse, '39' for default foreground color, '30' to '37' for foreground colors, '90' to '97' for 16-color mode foreground colors, '38;5;0' to '38;5;255' for 88-color and 256-color modes foreground colors, '49' for default background color, '40' to '47' for background colors, '100' to '107' for 16-color mode background colors, and '48;5;0' to '48;5;255' for 88-color and 256-color modes background colors.

Supported capabilities are as follows.

sl=
: SGR substring for whole selected lines (i.e., matching lines when the -v command-line option is omitted, or non-matching lines when -v is specified). If however the boolean 'rv' capability and the -v command-line option are both specified, it applies to context matching lines instead. The default is empty (i.e., the terminal's default color pair).

cx=
: SGR substring for whole context lines (i.e., non-matching lines when the -v command-line option is omitted, or matching lines when -v is specified). If however the boolean 'rv' capability and

the `-v` command-line option are both specified, it applies to selected non-matching lines instead. The default is empty (i.e., the terminal's default color pair).

`rv` Boolean value that reverses (swaps) the meanings of the '`sl=`' and '`cx=`' capabilities when the `-v` command-line option is specified. The default is false (i.e., the capability is omitted).

`mt=01;31` SGR substring for matching non-empty text in any matching line (i.e., a selected line when the `-v` command-line option is omitted, or a context line when `-v` is specified). Setting this is equivalent to setting both '`ms=`' and '`mc=`' at once to the same value. The default is a bold red text foreground over the current line background.

`ms=01;31` SGR substring for matching non-empty text in a selected line. (This is used only when the `-v` command-line option is omitted.) The effect of the '`sl=`' (or '`cx=`' if '`rv`') capability remains active when this takes effect. The default is a bold red text foreground over the current line background.

`mc=01;31` SGR substring for matching non-empty text in a context line. (This is used only when the `-v` command-line option is specified.) The effect of the '`cx=`' (or '`sl=`' if '`rv`') capability remains active when this takes effect. The default is a bold red text foreground over the current line background.

`fn=35` SGR substring for file names prefixing any content line. The default is a magenta text foreground over the terminal's default background.

`ln=32` SGR substring for line numbers prefixing any content line. The default is a green text foreground over the terminal's default background.

`bn=32` SGR substring for byte offsets prefixing any content line. The default is a green text foreground over the terminal's default background.

`se=36` SGR substring for separators that are inserted between selected line fields ('`:`'), between context line fields ('`-`'), and between groups of adjacent lines when nonzero context is specified ('`--`'). The default is a cyan text foreground over the terminal's default background.

`ne` Boolean value that prevents clearing to the end of line using Erase in Line (EL) to Right ('`\33[K`') each time a colorized item ends. This is needed on terminals on which EL is not supported. It is otherwise useful on terminals for which the `back_color_erase` (`bce`) boolean `terminfo` capability does not apply, when the chosen highlight colors do not affect the background, or when EL is too slow or causes too much flicker. The default is false (i.e., the capability is omitted).

Note that boolean capabilities have no '='... part. They are omitted (i.e., false) by default and become true when specified.

`LC_ALL`
`LC_COLLATE`
`LANG` These variables specify the locale for the `LC_COLLATE` category, which might affect how range expressions like '`a-z`' are interpreted.

`LC_ALL`
`LC_CTYPE`
`LANG` These variables specify the locale for the `LC_CTYPE` category, which determines the type of characters, e.g., which characters are whitespace. This category also determines the character encoding. See Section 3.8 [Character Encoding], page 19.

`LANGUAGE`
`LC_ALL`
`LC_MESSAGES`
`LANG` These variables specify the locale for the `LC_MESSAGES` category, which determines the language that `grep` uses for messages. The default '`C`' locale uses American English messages.

`POSIXLY_CORRECT`
If set, `grep` behaves as POSIX requires; otherwise, `grep` behaves more like other GNU programs. POSIX requires that options that follow file names must be treated as file names; by default, such options are permuted to the front of the operand list and are treated as options.

`TERM` This variable specifies the output terminal type, which can affect what the `--color` option does. See Section 2.1.3 [General Output Control], page 3.

The `GREP_OPTIONS` environment variable of `grep` 2.20 and earlier is no longer supported, as it caused problems when writing portable scripts. To make arbitrary changes to how `grep` works, you can use an alias or script instead. For example, if `grep` is in the directory '`/usr/bin`' you can prepend `$HOME/bin` to your `PATH` and create an executable script `$HOME/bin/grep` containing the following:

```
#! /bin/sh
export PATH=/usr/bin
exec grep --color=auto --devices=skip "$@"
```

2.3 Exit Status

Normally the exit status is 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred. However, if the `-q` or `--quiet` or `--silent` option is used and a line is selected, the exit status is 0 even if an error occurred. Other `grep` implementations may exit with status greater than 2 on error.

2.4 `grep` Programs

`grep` searches the named input files for lines containing a match to the given patterns. By default, `grep` prints the matching lines. A file named `-` stands for standard input. If

no input is specified, grep searches the working directory . if given a command-line option specifying recursion; otherwise, grep searches standard input. There are four major variants of grep, controlled by the following options.

-G
--basic-regexp
: Interpret patterns as basic regular expressions (BREs). This is the default.

-E
--extended-regexp
: Interpret patterns as extended regular expressions (EREs). (-E is specified by POSIX.)

-F
--fixed-strings
: Interpret patterns as fixed strings, not regular expressions. (-F is specified by POSIX.)

-P
--perl-regexp
: Interpret patterns as Perl-compatible regular expressions (PCREs). PCRE support is here to stay, but consider this option experimental when combined with the -z (--null-data) option, and note that 'grep -P' may warn of unimplemented features. See Section 2.1.7 [Other Options], page 9.

  For documentation, refer to https://www.pcre.org/, with these caveats:

  - '\d' matches only the ten ASCII digits (and '\D' matches the complement), regardless of locale. Use '\p{Nd}' to also match non-ASCII digits. (The behavior of '\d' and '\D' is unspecified after in-regexp directives like '(?aD)'.)
  - Although PCRE tracks the syntax and semantics of Perl's regular expressions, the match is not always exact. For example, Perl evolves and a Perl installation may predate or postdate the PCRE2 installation on the same host, or their Unicode versions may differ, or Perl and PCRE2 may disagree about an obscure construct.
  - By default, grep applies each regexp to a line at a time, so the '(?s)' directive (making '.' match line breaks) is generally ineffective. However, with -z (--null-data) it can work:

    ```
    $ printf 'a\nb\n' |grep -zP '(?s)a.b'
    a
    b
    ```

    But beware: with the -z (--null-data) and a file containing no NUL byte, grep must read the entire file into memory before processing any of it. Thus, it will exhaust memory and fail for some large files.

3 Regular Expressions

A *regular expression* is a pattern that describes a set of strings. Regular expressions are constructed analogously to arithmetic expressions, by using various operators to combine smaller expressions. `grep` understands three different versions of regular expression syntax: basic (BRE), extended (ERE), and Perl-compatible (PCRE). In GNU `grep`, basic and extended regular expressions are merely different notations for the same pattern-matching functionality. In other implementations, basic regular expressions are ordinarily less powerful than extended, though occasionally it is the other way around. The following description applies to extended regular expressions; differences for basic regular expressions are summarized afterwards. Perl-compatible regular expressions have different functionality, and are documented in the *pcre2syntax*(3) and *pcre2pattern*(3) manual pages, but work only if PCRE is available in the system.

3.1 Fundamental Structure

In regular expressions, the characters '`.?*+{|()[\^$`' are *special characters* and have uses described below. All other characters are *ordinary characters*, and each ordinary character is a regular expression that matches itself.

The period '`.`' matches any single character. It is unspecified whether '`.`' matches an encoding error.

A regular expression may be followed by one of several repetition operators; the operators beginning with '`{`' are called *interval expressions*.

'`?`' The preceding item is optional and is matched at most once.

'`*`' The preceding item is matched zero or more times.

'`+`' The preceding item is matched one or more times.

'`{`*n*`}`' The preceding item is matched exactly *n* times.

'`{`*n*`,}`' The preceding item is matched *n* or more times.

'`{,`*m*`}`' The preceding item is matched at most *m* times. This is a GNU extension.

'`{`*n*`,`*m*`}`' The preceding item is matched at least *n* times, but not more than *m* times.

The empty regular expression matches the empty string. Two regular expressions may be concatenated; the resulting regular expression matches any string formed by concatenating two substrings that respectively match the concatenated expressions.

Two regular expressions may be joined by the infix operator '`|`'. The resulting regular expression matches any string matching either of the two expressions, which are called *alternatives*.

Repetition takes precedence over concatenation, which in turn takes precedence over alternation. A whole expression may be enclosed in parentheses to override these precedence rules and form a subexpression. An unmatched '`)`' matches just itself.

Not every character string is a valid regular expression. See Section 3.7 [Problematic Expressions], page 18.

3.2 Character Classes and Bracket Expressions

A *bracket expression* is a list of characters enclosed by '`[`' and '`]`'. It matches any single character in that list. If the first character of the list is the caret '`^`', then it matches any character **not** in the list, and it is unspecified whether it matches an encoding error. For example, the regular expression '`[0123456789]`' matches any single digit, whereas '`[^()]`' matches any single character that is not an opening or closing parenthesis, and might or might not match an encoding error.

Within a bracket expression, a *range expression* consists of two characters separated by a hyphen. It matches any single character that sorts between the two characters, inclusive. In the default C locale, the sorting sequence is the native character order; for example, '`[a-d]`' is equivalent to '`[abcd]`'. In other locales, the sorting sequence is not specified, and '`[a-d]`' might be equivalent to '`[abcd]`' or to '`[aBbCcDd]`', or it might fail to match any character, or the set of characters that it matches might be erratic, or it might be invalid. To obtain the traditional interpretation of bracket expressions, you can use the '`C`' locale by setting the `LC_ALL` environment variable to the value '`C`'.

Finally, certain named classes of characters are predefined within bracket expressions, as follows. Their interpretation depends on the `LC_CTYPE` locale; for example, '`[[:alnum:]]`' means the character class of numbers and letters in the current locale.

'`[:alnum:]`'
: Alphanumeric characters: '`[:alpha:]`' and '`[:digit:]`'; in the '`C`' locale and ASCII character encoding, this is the same as '`[0-9A-Za-z]`'.

'`[:alpha:]`'
: Alphabetic characters: '`[:lower:]`' and '`[:upper:]`'; in the '`C`' locale and ASCII character encoding, this is the same as '`[A-Za-z]`'.

'`[:blank:]`'
: Blank characters: space and tab.

'`[:cntrl:]`'
: Control characters. In ASCII, these characters have octal codes 000 through 037, and 177 (DEL). In other character sets, these are the equivalent characters, if any.

'`[:digit:]`'
: Digits: `0 1 2 3 4 5 6 7 8 9`.

'`[:graph:]`'
: Graphical characters: '`[:alnum:]`' and '`[:punct:]`'.

'`[:lower:]`'
: Lower-case letters; in the '`C`' locale and ASCII character encoding, this is `a b c d e f g h i j k l m n o p q r s t u v w x y z`.

'`[:print:]`'
: Printable characters: '`[:alnum:]`', '`[:punct:]`', and space.

'`[:punct:]`'
: Punctuation characters; in the '`C`' locale and ASCII character encoding, this is `! " # $ % & ' ( ) * + , - . / : ; < = > ? @ [ \ ] ^ _ ` { | } ~`.

‘`[:space:]`’
: Space characters: in the ‘`C`’ locale, this is tab, newline, vertical tab, form feed, carriage return, and space. See Chapter 4 [Usage], page 21, for more discussion of matching newlines.

‘`[:upper:]`’
: Upper-case letters: in the ‘`C`’ locale and ASCII character encoding, this is `A B C D E F G H I J K L M N O P Q R S T U V W X Y Z`.

‘`[:xdigit:]`’
: Hexadecimal digits: `0 1 2 3 4 5 6 7 8 9 A B C D E F a b c d e f`.

Note that the brackets in these class names are part of the symbolic names, and must be included in addition to the brackets delimiting the bracket expression.

If you mistakenly omit the outer brackets, and search for say, ‘`[:upper:]`’, GNU `grep` prints a diagnostic and exits with status 2, on the assumption that you did not intend to search for the regular expression ‘`[:epru]`’.

Special characters lose their special meaning inside bracket expressions.

‘`]`’
: ends the bracket expression if it’s not the first list item. So, if you want to make the ‘`]`’ character a list item, you must put it first.

‘`[.`’
: represents the open collating symbol.

‘`.]`’
: represents the close collating symbol.

‘`[=`’
: represents the open equivalence class.

‘`=]`’
: represents the close equivalence class.

‘`[:`’
: represents the open character class symbol, and should be followed by a valid character class name.

‘`:]`’
: represents the close character class symbol.

‘`-`’
: represents the range if it’s not first or last in a list or the ending point of a range. To make the ‘`-`’ a list item, it is best to put it last.

‘`^`’
: represents the characters not in the list. If you want to make the ‘`^`’ character a list item, place it anywhere but first.

3.3 Special Backslash Expressions

The ‘`\`’ character followed by a special character is a regular expression that matches the special character. The ‘`\`’ character, when followed by certain ordinary characters, takes a special meaning:

‘`\b`’
: Match the empty string at the edge of a word.

‘`\B`’
: Match the empty string provided it’s not at the edge of a word.

‘`\<`’
: Match the empty string at the beginning of a word.

‘`\>`’
: Match the empty string at the end of a word.

‘`\w`’
: Match word constituent, it is a synonym for ‘`[_[:alnum:]]`’.

‘`\W`’ Match non-word constituent, it is a synonym for ‘`[^_[:alnum:]]`’.

‘`\s`’ Match whitespace, it is a synonym for ‘`[[:space:]]`’.

‘`\S`’ Match non-whitespace, it is a synonym for ‘`[^[:space:]]`’.

‘`\]`’ Match ‘`]`’.

‘`\}`’ Match ‘`}`’.

For example, ‘`\brat\b`’ matches the separate word ‘`rat`’, ‘`\Brat\B`’ matches ‘`crate`’ but not ‘`furry rat`’.

The behavior of `grep` is unspecified if a unescaped backslash is not followed by a special character, a nonzero digit, or a character in the above list. Although `grep` might issue a diagnostic and/or give the backslash an interpretation now, its behavior may change if the syntax of regular expressions is extended in future versions.

3.4 Anchoring

The caret ‘`^`’ and the dollar sign ‘`$`’ are special characters that respectively match the empty string at the beginning and end of a line. They are termed *anchors*, since they force the match to be “anchored” to beginning or end of a line, respectively.

3.5 Back-references and Subexpressions

The back-reference ‘`\`*n*’, where *n* is a single nonzero digit, matches the substring previously matched by the *n*th parenthesized subexpression of the regular expression. For example, ‘`(a)\1`’ matches ‘`aa`’. If the parenthesized subexpression does not participate in the match, the back-reference makes the whole match fail; for example, ‘`(a)*\1`’ fails to match ‘`a`’. If the parenthesized subexpression matches more than one substring, the back-reference refers to the last matched substring; for example, ‘`^(ab*)*\1$`’ matches ‘`ababbabb`’ but not ‘`ababbab`’. When multiple regular expressions are given with `-e` or from a file (‘`-f` *`file`*’), back-references are local to each expression.

See Section 6.1 [Known Bugs], page 27, for some known problems with back-references.

3.6 Basic vs Extended Regular Expressions

Basic regular expressions differ from extended regular expressions in the following ways:

- The characters ‘`?`’, ‘`+`’, ‘`{`’, ‘`|`’, ‘`(`’, and ‘`)`’ lose their special meaning; instead use the backslashed versions ‘`\?`’, ‘`\+`’, ‘`\{`’, ‘`\|`’, ‘`\(`’, and ‘`\)`’. Also, a backslash is needed before an interval expression’s closing ‘`}`’.
- An unmatched ‘`\)`’ is invalid.
- If an unescaped ‘`^`’ appears neither first, nor directly after ‘`\(`’ or ‘`\|`’, it is treated like an ordinary character and is not an anchor.
- If an unescaped ‘`$`’ appears neither last, nor directly before ‘`\|`’ or ‘`\)`’, it is treated like an ordinary character and is not an anchor.
- If an unescaped ‘`*`’ appears first, or appears directly after ‘`\(`’ or ‘`\|`’ or anchoring ‘`^`’, it is treated like an ordinary character and is not a repetition operator.

3.7 Problematic Regular Expressions

Some strings are *invalid regular expressions* and cause `grep` to issue a diagnostic and fail. For example, '`xy\1`' is invalid because there is no parenthesized subexpression for the back-reference '`\1`' to refer to.

Also, some regular expressions have *unspecified behavior* and should be avoided even if `grep` does not currently diagnose them. For example, '`xy\0`' has unspecified behavior because '`0`' is not a special character and '`\0`' is not a special backslash expression (see Section 3.3 [Special Backslash Expressions], page 16). Unspecified behavior can be particularly problematic because the set of matched strings might be only partially specified, or not be specified at all, or the expression might even be invalid.

The following regular expression constructs are invalid on all platforms conforming to POSIX, so portable scripts can assume that `grep` rejects these constructs:

- A basic regular expression containing a back-reference '`\n`' preceded by fewer than *n* closing parentheses. For example, '`\(a\)\2`' is invalid.
- A bracket expression containing '`[:`' that does not start a character class; and similarly for '`[=`' and '`[.`'. For example, '`[a[:b]`' and '`[a[:ouch:]b]`' are invalid.

GNU `grep` treats the following constructs as invalid. However, other `grep` implementations might allow them, so portable scripts should not rely on their being invalid:

- Unescaped '`\`' at the end of a regular expression.
- Unescaped '`[`' that does not start a bracket expression.
- A '`\{`' in a basic regular expression that does not start an interval expression.
- A basic regular expression with unbalanced '`\(`' or '`\)`', or an extended regular expression with unbalanced '`(`'.
- In the POSIX locale, a range expression like '`z-a`' that represents zero elements. A non-GNU `grep` might treat it as a valid range that never matches.
- An interval expression with a repetition count greater than 32767. (The portable POSIX limit is 255, and even interval expressions with smaller counts can be impractically slow on all known implementations.)
- A bracket expression that contains at least three elements, the first and last of which are both '`:`', or both '`.`', or both '`=`'. For example, a non-GNU `grep` might treat '`[:alpha:]`' like '`[[:alpha:]]`', or like '`[:ahlp]`'.

The following constructs have well-defined behavior in GNU `grep`. However, they have unspecified behavior elsewhere, so portable scripts should avoid them:

- Special backslash expressions like '`\b`', '`\<`', and '`\]`'. See Section 3.3 [Special Backslash Expressions], page 16.
- A basic regular expression that uses '`\?`', '`\+`', or '`\|`'.
- An extended regular expression that uses back-references.
- An empty regular expression, subexpression, or alternative. For example, '`(a|bc|)`' is not portable; a portable equivalent is '`(a|bc)?`'.
- In a basic regular expression, an anchoring '`^`' that appears directly after '`\(`', or an anchoring '`$`' that appears directly before '`\)`'.

- In a basic regular expression, a repetition operator that directly follows another repetition operator.
- In an extended regular expression, unescaped '`{`' that does not begin a valid interval expression. GNU `grep` treats the '`{`' as an ordinary character.
- A null character or an encoding error in either pattern or input data. See Section 3.8 [Character Encoding], page 19.
- An input file that ends in a non-newline character, where GNU `grep` silently supplies a newline.

The following constructs have unspecified behavior, in both GNU and other `grep` implementations. Scripts should avoid them whenever possible.

- A backslash escaping an ordinary character, unless it is a back-reference like '`\1`' or a special backslash expression like '`\<`' or '`\b`'. See Section 3.3 [Special Backslash Expressions], page 16. For example, '`\x`' has unspecified behavior now, and a future version of `grep` might specify '`\x`' to have a new behavior.
- A repetition operator that appears directly after an anchor, or at the start of a complete regular expression, parenthesized subexpression, or alternative. For example, '`+|^*(+a|?-b)`' has unspecified behavior, whereas '`\+|^\*(\+a|\?-b)`' is portable.
- A range expression outside the POSIX locale. For example, in some locales '`[a-z]`' might match some characters that are not lowercase letters, or might not match some lowercase letters, or might be invalid. With GNU `grep` it is not documented whether these range expressions use native code points, or use the collating sequence specified by the `LC_COLLATE` category, or have some other interpretation. Outside the POSIX locale, it is portable to use '`[[:lower:]]`' to match a lower-case letter, or '`[abcdefghijklmnopqrstuvwxyz]`' to match an ASCII lower-case letter.

3.8 Character Encoding

The `LC_CTYPE` locale specifies the encoding of characters in patterns and data, that is, whether text is encoded in UTF-8, ASCII, or some other encoding. See Section 2.2 [Environment Variables], page 9.

In the '`C`' or '`POSIX`' locale, every character is encoded as a single byte and every byte is a valid character. In more-complex encodings such as UTF-8, a sequence of multiple bytes may be needed to represent a character, and some bytes may be encoding errors that do not contribute to the representation of any character. POSIX does not specify the behavior of `grep` when patterns or input data contain encoding errors or null characters, so portable scripts should avoid such usage. As an extension to POSIX, GNU `grep` treats null characters like any other character. However, unless the `-a` (`--binary-files=text`) option is used, the presence of null characters in input or of encoding errors in output causes GNU `grep` to treat the file as binary and suppress details about matches. See Section 2.1.6 [File and Directory Selection], page 7.

Regardless of locale, the 103 characters in the POSIX Portable Character Set (a subset of ASCII) are always encoded as a single byte, and the 128 ASCII characters have their usual single-byte encodings on all but oddball platforms.

3.9 Matching Non-ASCII and Non-printable Characters

In a regular expression, non-ASCII and non-printable characters other than newline are not special, and represent themselves. For example, in a locale using UTF-8 the command '`grep 'Λ ω'`' (where the white space between 'Λ' and the 'ω' is a tab character) searches for 'Λ' (Unicode character U+039B GREEK CAPITAL LETTER LAMBDA), followed by a tab (U+0009 TAB), followed by 'ω' (U+03C9 GREEK SMALL LETTER OMEGA).

Suppose you want to limit your pattern to only printable characters (or even only printable ASCII characters) to keep your script readable or portable, but you also want to match specific non-ASCII or non-null non-printable characters. If you are using the `-P` (`--perl-regexp`) option, PCREs give you several ways to do this. Otherwise, if you are using Bash, the GNU project's shell, you can represent these characters via ANSI-C quoting. For example, the Bash commands '`grep $'Λ\tω'`' and '`grep $'\u039B\t\u03C9'`' both search for the same three-character string 'Λ ω' mentioned earlier. However, because Bash translates ANSI-C quoting before `grep` sees the pattern, this technique should not be used to match printable ASCII characters; for example, '`grep $'\u005E'`' is equivalent to '`grep '^'`' and matches any line, not just lines containing the character '^' (U+005E CIRCUMFLEX ACCENT).

Since PCREs and ANSI-C quoting are GNU extensions to POSIX, portable shell scripts written in ASCII should use other methods to match specific non-ASCII characters. For example, in a UTF-8 locale the command '`grep "$(printf '\316\233\t\317\211\n')"`' is a portable albeit hard-to-read alternative to Bash's '`grep $'Λ\tω'`'. However, none of these techniques will let you put a null character directly into a command-line pattern; null characters can appear only in a pattern specified via the `-f` (`--file`) option.

4 Usage

Here is an example command that invokes GNU `grep`:

```
grep -i 'hello.*world' menu.h main.c
```

This lists all lines in the files `menu.h` and `main.c` that contain the string '`hello`' followed by the string '`world`'; this is because '`.*`' matches zero or more characters within a line. See Chapter 3 [Regular Expressions], page 14. The `-i` option causes `grep` to ignore case, causing it to match the line '`Hello, world!`', which it would not otherwise match.

Here is a more complex example, showing the location and contents of any line containing '`f`' and ending in '`.c`', within all files in the current directory whose names start with non-'`.`', contain '`g`', and end in '`.h`'. The `-n` option outputs line numbers, the `--` argument treats any later arguments as file names not options even if `*g*.h` expands to a file name that starts with '`-`', and the empty file `/dev/null` causes file names to be output even if only one file name happens to be of the form '`*g*.h`'.

```
grep -n -- 'f.*\.c$' *g*.h /dev/null
```

Note that the regular expression syntax used in the pattern differs from the globbing syntax that the shell uses to match file names.

See Chapter 2 [Invoking], page 2, for more details about how to invoke `grep`.

Here are some common questions and answers about `grep` usage.

1. How can I list just the names of matching files?

   ```
   grep -l 'main' test-*.c
   ```

   lists names of '`test-*.c`' files in the current directory whose contents mention '`main`'.

2. How do I search directories recursively?

   ```
   grep -r 'hello' /home/gigi
   ```

   searches for '`hello`' in all files under the `/home/gigi` directory. For more control over which files are searched, use `find` and `grep`. For example, the following command searches only C files:

   ```
   find /home/gigi -name '*.c' ! -type d \
     -exec grep -H 'hello' '{}' +
   ```

   This differs from the command:

   ```
   grep -H 'hello' /home/gigi/*.c
   ```

   which merely looks for '`hello`' in non-hidden C files in `/home/gigi` whose names end in '`.c`'. The `find` command line above is more similar to the command:

   ```
   grep -r --include='*.c' 'hello' /home/gigi
   ```

3. What if a pattern or file has a leading '`-`'? For example:

   ```
   grep "$pattern" *
   ```

   can behave unexpectedly if the value of '`pattern`' begins with '`-`', or if the '`*`' expands to a file name with leading '`-`'. To avoid the problem, you can use `-e` for patterns and leading '`./`' for files:

   ```
   grep -e "$pattern" ./*
   ```

   searches for all lines matching the pattern in all the working directory's files whose names do not begin with '`.`'. Without the `-e`, `grep` might treat the pattern as an

option if it begins with '`-`'. Without the '`./`', there might be similar problems with file names beginning with '`-`'.

Alternatively, you can use '`--`' before the pattern and file names:

```
grep -- "$pattern" *
```

This also fixes the problem, except that if there is a file named '`-`', `grep` misinterprets the '`-`' as standard input.

4. Suppose I want to search for a whole word, not a part of a word?

   ```
   grep -w 'hello' test*.log
   ```

   searches only for instances of '`hello`' that are entire words; it does not match '`Othello`'. For more control, use '`\<`' and '`\>`' to match the start and end of words. For example:

   ```
   grep 'hello\>' test*.log
   ```

   searches only for words ending in '`hello`', so it matches the word '`Othello`'.

5. How do I output context around the matching lines?

   ```
   grep -C 2 'hello' test*.log
   ```

   prints two lines of context around each matching line.

6. How do I force `grep` to print the name of the file?

   Append `/dev/null`:

   ```
   grep 'eli' /etc/passwd /dev/null
   ```

   gets you:

   ```
   /etc/passwd:eli:x:2098:1000:Eli Smith:/home/eli:/bin/bash
   ```

   Alternatively, use `-H`, which is a GNU extension:

   ```
   grep -H 'eli' /etc/passwd
   ```

7. Why do people use strange regular expressions on `ps` output?

   ```
   ps -ef | grep '[c]ron'
   ```

   If the pattern had been written without the square brackets, it would have matched not only the `ps` output line for `cron`, but also the `ps` output line for `grep`. Note that on some platforms, `ps` limits the output to the width of the screen; `grep` does not have any limit on the length of a line except the available memory.

8. Why does `grep` report "Binary file matches"?

   If `grep` listed all matching "lines" from a binary file, it would probably generate output that is not useful, and it might even muck up your display. So GNU `grep` suppresses output from files that appear to be binary files. To force GNU `grep` to output lines even from files that appear to be binary, use the `-a` or '`--binary-files=text`' option. To eliminate the "Binary file matches" messages, use the `-I` or '`--binary-files=without-match`' option.

9. Why doesn't '`grep -lv`' print non-matching file names?

   '`grep -lv`' lists the names of all files containing one or more lines that do not match. To list the names of all files that contain no matching lines, use the `-L` or `--files-without-match` option.

10. I can do "OR" with '`|`', but what about "AND"?

    ```
    grep 'paul' /etc/motd | grep 'franc,ois'
    ```

    finds all lines that contain both '`paul`' and '`franc,ois`'.

11. Why does the empty pattern match every input line?

    The `grep` command searches for lines that contain strings that match a pattern. Every line contains the empty string, so an empty pattern causes `grep` to find a match on each line. It is not the only such pattern: '`^`', '`$`', and many other patterns cause `grep` to match every line.

    To match empty lines, use the pattern '`^$`'. To match blank lines, use the pattern '`^[[:blank:]]*$`'. To match no lines at all, use an extended regular expression like '`a^`' or '`$a`'. To match every line, a portable script should use a pattern like '`^`' instead of the empty pattern, as POSIX does not specify the behavior of the empty pattern.

12. How can I search in both standard input and in files?

    Use the special file name '`-`':

    ```
    cat /etc/passwd | grep 'alain' - /etc/motd
    ```

13. Why can't I combine the shell's '`set -e`' with `grep`?

    The `grep` command follows the convention of programs like `cmp` and `diff` where an exit status of 1 is not an error. The shell command '`set -e`' causes the shell to exit if any subcommand exits with nonzero status, and this will cause the shell to exit merely because `grep` selected no lines, which is ordinarily not what you want.

    There is a related problem with Bash's `set -e -o pipefail`. Since `grep` does not always read all its input, a command outputting to a pipe read by `grep` can fail when `grep` exits before reading all its input, and the command's failure can cause Bash to exit.

14. Why is this back-reference failing?

    ```
    echo 'ba' | grep -E '(a)\1|b\1'
    ```

    This outputs an error message, because the second '`\1`' has nothing to refer back to, meaning it will never match anything.

15. How can I match across lines?

    Standard grep cannot do this, as it is fundamentally line-based. Therefore, merely using the `[:space:]` character class does not match newlines in the way you might expect.

    With the GNU `grep` option `-z` (`--null-data`), each input and output "line" is null-terminated; see Section 2.1.7 [Other Options], page 9. Thus, you can match newlines in the input, but typically if there is a match the entire input is output, so this usage is often combined with output-suppressing options like `-q`, e.g.:

    ```
    printf 'foo\nbar\n' | grep -z -q 'foo[[:space:]]\+bar'
    ```

    If this does not suffice, you can transform the input before giving it to `grep`, or turn to `awk`, `sed`, `perl`, or many other utilities that are designed to operate across lines.

16. What do `grep`, `-E`, and `-F` stand for?

    The name `grep` comes from the way line editing was done on Unix. For example, `ed` uses the following syntax to print a list of matching lines on the screen:

    ```
    global/regular expression/print
    g/re/p
    ```

    The `-E` option stands for Extended `grep`. The `-F` option stands for Fixed `grep`;

17. What happened to `egrep` and `fgrep`?

    7th Edition Unix had commands `egrep` and `fgrep` that were the counterparts of the modern '`grep -E`' and '`grep -F`'. Although breaking up `grep` into three programs was perhaps useful on the small computers of the 1970s, `egrep` and `fgrep` were deemed obsolescent by POSIX in 1992, removed from POSIX in 2001, deprecated by GNU Grep 2.5.3 in 2007, and changed to issue obsolescence warnings by GNU Grep 3.8 in 2022; eventually, they are planned to be removed entirely.

    If you prefer the old names, you can use your own substitutes, such as a shell script named `egrep` with the following contents:

    ```
    #!/bin/sh
    exec grep -E "$@"
    ```

The operator of an MMC Site may republish an MMC contained in the site under CC-BY-SA on the same site at any time before August 1, 2009, provided the MMC is eligible for relicensing.