1 Finding Information with Regular Expressions and the grep Command

This chapter describes regular expressions and how to use them. Regular expressions are most commonly used in the context of pattern matching with the `grep` command, but they are also used with virtually all text-processing or filtering utilities and commands. A more thorough discussion of the `grep` command follows the exposition of regular expressions.

1.1 Forming Regular Expressions

A regular expression specifies a set of strings to be matched. It contains ordinary text characters and **operator** characters. Ordinary characters match the corresponding characters in the strings being compared. Operator characters specify repetitions, choices, and other features. Regular expressions fall into two groups:

- Basic regular expressions
- Extended regular expressions

Section 1.1.1 and Section 1.1.2 describe the two types of regular expressions. In addition to the constructs described in these two sections, there are three special expression types related to character classes, collating sequences, and equivalence classes. See Section 1.1.6 for more information on these classes. The order of precedence of the regular expression operators discussed in these three sections is as follows:

1. Collation-related bracket symbols: `[= =]`, `[. .]`, and `[: :]`
2. Escaped operator characters: `\`*`char`*
3. Bracket expressions: `[`*`expr`*`]`
4. Subexpressions and back-reference expressions: `\(`*`expr`*`\)`, `\`*`n`* in basic regular expressions; `(`*`expr`*`)` only in extended regular expressions
5. Duplication: `*`, `\{`*`i`*`\}`, `\{`*`i`*`,\}`, `\{`*`i`*`,`*`j`*`\}` in basic regular expressions; `*`, `?`, `+`, `{`*`i`*`}`, `{`*`i`*`,}`, `{`*`i`*`,`*`j`*`}` in extended regular expressions
6. Concatenation
7. Anchoring: `^`, `$`
8. Alternation in extended regular expressions: `|`

1.1.1 Basic Regular Expressions

Basic regular expressions are built by concatenating simpler basic regular expressions. The letters of the alphabet are ordinary characters. An ordinary character is an expression that always matches itself and nothing else. (Usually, digits are also ordinary characters, but a digit preceded by a backslash forms a "back-reference expression"; see Table 1-1.) For example, the expression `rabbit` matches the string `rabbit`, and the expression `a57D` matches the string `a57D`.

Ordinary characters and operator characters together make up the set of simple basic regular expressions. You can concatenate any number or combination of simple expressions to create a compound expression that will match any sequence of characters that corresponds to the concatenated simple expressions. Table 1-1 describes the rules for creating basic regular expressions.

The sections following Section 1.1.2 provide further explanation of some of the expressions listed in Table 1-1 and Table 1-2.

Table 1-1: Rules for Basic Regular Expressions

Expression	Name	Description
letters, numbers, most punctuation	Ordinary character	Matches itself.
.	Period (dot)	Matches any single character except the newline character.
*	Asterisk	Matches any number of occurrences of the preceding simple expression, including none.
`\{`*i*`,`*j*`\}`	Interval expression	Matches a more restricted number of instances of the preceding simple expression; for example, `ab\{3\}c` matches only `abbbc`, while `ab\{2,3\}c` matches `abbc` or `abbbc`, but not `abc` or `abbbbc`.
`\(`*expr*`\)`	Subexpression (hold delimiters)	Matches *expr*, causing basic regular expression operators to treat it as a unit; for example, `a\(bc\)\{2,3\}d` matches `abcbcd` or `abcbcbcd` but not `abcd` or `abcbcbcbcd`. Additionally, the subexpression is saved into a numbered holding space (up to nine per expression) for reuse later in the expression to specify another match on the same subexpression.
`\`*n*	Back-reference expression	Repeats the contents of the *n*th subexpression in the regular expression.
`[`*chars*`]`	Bracket expression	Matches a single instance of any one of the characters within the brackets. Ranges of characters can be abbreviated by using a hyphen. For example, `[0-9a-z]` matches any single digit or lowercase letter. Within brackets, all characters are

Expression	Name	Description
		ordinary characters except the hyphen (when used in a range abbreviation) and the circumflex (when used as the first character inside the brackets).
^	Circumflex	When used at the beginning of a regular expression (or a subexpression), matches the beginning of a line ("anchors" the expression to the beginning of the line). When used as the first character inside brackets, excludes the bracketed characters from being matched. Otherwise, has no special properties.
$	Dollar sign	When used at the end of a regular expression, matches the end of a line ("anchors" the expression to the end of the line). Otherwise, has no special properties.
char	Backslash	Except within a bracket expression, escapes the next character to permit matching on explicit instances of characters that are normally basic regular expression operators.
expr expr ...	Concatenation	Matches any string that matches all of the concatenated expressions in sequence.

1.1.2 Extended Regular Expressions

In general, extended regular expressions are like the basic regular expressions described in Section 1.1.1. However, extended regular expressions comprise a larger set that is used by certain programs, such as `awk`, that can perform more powerful file-manipulation and filtering operations than programs such as `grep` (when used without its `-E` flag) or `sed`. It is better, then, to consider extended regular expressions separately from basic regular expressions despite the fact that the two types of expressions share many constructs. Table 1-2 lists the rules for forming extended regular expressions; note that constructs that are shared between basic and extended regular expressions are listed both here and in Table 1-1.

Table 1-2: Rules for Extended Regular Expressions

Expression	Name	Description
letters, numbers, most punctuation	Ordinary character	Matches itself.
.	Period (dot)	Matches any single character except the newline character.

Expression	Name	Description
*	Asterisk	Matches any number of occurrences of the preceding simple expression, including none.
?	Question mark	Matches zero or one occurrence of the preceding simple expression.
+	Plus sign	Matches one or more occurrences of the preceding simple expression.
{*i*,*j*}	Interval expression	Matches a more restricted number of instances of the preceding simple expression; for example, `ab{3}c` matches only `abbbc`, while `ab{2,3}c` matches `abbc` or `abbbc`, but not `abc` or `abbbbc`. Note that basic regular expression interval expressions are delimited by escaped braces. To match a literal expression that has the form of an interval expression using an extended regular expression, escape the left brace. For example, `\{2,3}` matches the explicit string `{2,3}`.
(*expr*)	Subexpression	Matches *expr*, causing extended regular expression operators to treat it as a unit; for example, `a(bc)?d` matches `ad` or `abcd` but not `abcbcd`, `abcbcbcd`, or other similar strings. Note that basic regular expression subexpresions are delimited by escaped parentheses. To match a literal parenthesized expression using an extended regular expression, escape the left parenthesis. For example, `\(abc)` matches the explicit string `(abc)`.
[*chars*]	Bracket expression	Matches a single instance of any one of the characters within the brackets. Ranges of characters can be abbreviated by using a hyphen. For example, `[0-9a-z]` matches any single digit or lowercase letter. Within brackets, all characters are ordinary characters except the hyphen (when used in a range abbreviation) and the circumflex (when used as the first character inside the brackets).
^	Circumflex	When used at the beginning of an expression (or a subexpression), matches the beginning of a line ("anchors" the expression to the beginning of the line). When used as the first character inside brackets, excludes the bracketed characters from being matched.

Expression	Name	Description
		Otherwise, has no special properties.
$	Dollar sign	When used at the end of an expression, matches the end of a line ("anchors" the expression to the end of the line). Otherwise, has no special properties.
\char	Backslash	Except within a bracket expression, escapes the next character to permit matching on explicit instances of characters that are normally extended regular expression operators.
expr expr ...	Concatenation	Matches any string that matches all of the concatenated expressions in sequence.
expr|*expr ...*	Vertical bar (alternation)	Separates multiple extended regular expressions; matches any of the bar-separated expressions.

1.1.3 Matching Multiple Occurrences of a Regular Expression

An asterisk (*) acts on the simple regular expression immediately preceding it, causing that expression to match any number of occurrences of a matching pattern, even none. When an asterisk follows a period, the combination indicates a match on any sequence of characters, even none. A period and an asterisk always match as much text as possible; for example:

```
% echo "A B C D" | sed 's/^.* /E/'
ED
```

The sed stream editor command in this example indicates that sed is to match the expression between the first and second slashes and replace the matching pattern with the string between the second and third slashes. This regular expression will match any string that starts at the beginning of the line, contains any sequence of characters, and ends in a space. Nominally, the string "A " satisfies this expression; but the longest matching pattern is "A B C ", so sed replaces "A B C " with "E" to yield ED as the output. See Chapter 3 for a discussion of the sed stream editor.

An asterisk matches any number of instances of the preceding regular expression (both basic and extended). To limit the number of instances that a particular extended regular expression will match, use a plus sign (+) or a question mark (?). The plus sign requires at least one instance of its matching pattern. The question mark refuses to accept more than one instance. The following chart illustrates the matching characteristics of the asterisk, plus sign, and question mark:

Regular Expression	Matching Strings		
ab?c	ac	abc	
ab*c	ac	abc	abbc, abbbc, ...

ab+c		abc	abbc, abbbc, ...

You can also specify more restrictive numbers of instances of the desired regular expression with an interval expression. The following list illustrates the various forms of interval expressions in basic regular expressions:

- *expr*\{*i*\} Matches exactly *i* instances of anything `expr` matches. For example, `ab\{3\}c` matches `abbbc` but does not match either `abbc` or `abbbbc`.
- `\{`*i*`,\}` Matches at least *i* instances. For example, `ab\{3,\}c` matches `abbbc`, `abbbbc`, and so on, but not `ac`, `abc`, or `abbc`.
- `\{`*i*`,`*j*`\}` Matches any number of instances from *i* to *j*, inclusive. For example, `ab\{2,4\}c` matches `abbc`, `abbbc`, or `abbbbc` but not `abc` or `abbbbbc`. You can use 0 (zero) for *i*.

For extended regular expressions, omit the backslashes, making the previous examples `ab{3}c`, `ab{3,}c`, and `ab{2,4}c`.

Using the subexpression delimiters, you can save up to nine basic regular expression subexpression patterns on a line. Counting from left to right on the line, the first pattern saved is placed in the first holding space, the second pattern is placed in the second holding space, and so on.

The back-reference character sequence `\`*n* (where *n* is a digit from 1 to 9) matches the *n*th saved pattern. Consider the following basic regular expression:

```
\(A\)\(B\)C\2\1
```

This expression matches the string `ABCBA`. You can nest patterns to be saved in holding spaces. Whether the enclosed patterns are nested or in a series, *n* refers to the *n*th occurrence, counting from the left, of the subexpression delimiters. You can also use `\`*n* back-reference expressions in replacement strings as well as address patterns for editors such as `ed` and `sed`. Extended regular expressions do not support back-referencing.

1.1.4 Matching Only Selected Characters

A period in an expression matches any character except the newline character. To restrict the characters to be matched, place the desired characters inside brackets (`[ ]`). Each string of bracketed characters is a single-character expression that matches any one of the bracketed characters. Except for the circumflex (`^`), regular expression operators within brackets are interpreted literally, without special meaning. The circumflex excludes the bracketed characters if it is the first character in the brackets; otherwise, it has no special meaning.

When you specify a range of characters with a hyphen (for example, `[a-z]`), the characters that fall within the range are determined by the current collating sequence defined by the current setting of the `LC_CTYPE` environment variable. (See the discussion on using internationalization features in the *Command and Shell User's Guide* for more information on collating sequences.) The hyphen has no special meaning if it is the first or last character in a bracketed string or in a range

expression in a bracketed string, or if it immediately follows a circumflex that is the first character in a bracketed string. To include a right bracket in a bracket expression, place it first or after the initial circumflex.

You can use the `grep` command's `-i` flag to perform a case insensitive match. (The `-y` flag is an exact synonym for `-i`.) To create an expression that is not case sensitive for other utilities, or to form an expression that is only partially case insensitive, use a bracket expression consisting of just the uppercase and lowercase versions of the character you want. For example:

```
% grep '[Jj]ones' group-list
```

1.1.5 Specifying Multiple Regular Expressions

Some utilities, such as `grep` (with its `-E` flag) and `awk`, permit you to specify multiple alternative extended regular expressions simultaneously by separating the individual expressions with a vertical bar. For example:

```
% awk '/[Bb]lack|[Ww]hite/ {print NR ":", $0}' .Xdefaults
55: sm.pointer_foreground:  black
56: sm.pointer_background:  white
```

1.1.6 Special Collating Considerations in Regular Expressions

Bracket expressions can include three special types of expressions called classes:

- Character class

  Specifies a general type of character, such as uppercase letters.

- Collating-symbol class

  In internationalized usages, specifies multicharacter strings that sort as single characters.

- Equivalence class

  In internationalized usages, specifies collections of characters that have the same primary sort value.

Note that when not used within a bracket expression, all of the constructs described in this section are interpreted literally as the explicit sequences of characters that make them up.

A character class name enclosed in bracket-colon delimiters, `[:` and `:]`, matches any of the set of characters in the named class. Members of each of the sets are determined by the current setting of the `LC_CTYPE` environment variable. The supported classes are `alnum`, `alpha`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, and `xdigit`. For example, `[[:lower:]]` matches any lowercase letter in the current locale.

Some collating sequences include multicharacter strings that must be sorted as if they were single characters. For example, in Hungarian, the strings `cs`, `dz`, and others are each collating symbols. (The Hungarian primary sort order is `a`, `á`, `b`, `c`, `cs`, `d`, `dz`, `e`, ...). These special strings are called **collating symbols**, and they are

indicated by being enclosed within bracket-period delimiters, `[.` and `.]`. The bracket-period delimiters in the regular expression syntax distinguish multicharacter collating elements from a list of the individual characters that make up the element. When using Hungarian collation rules, for example, `[[.cs.]]` is treated as an expression matching the sequence `cs`, while `[cs]` is treated as an expression matching `c` or `s`. In addition, `[a-[.cs.]]` matches `a`, `á`, `b`, `c`, and `cs`.

A collating sequence can define equivalence classes for characters. An equivalence class is a set of collating elements that all sort to the same primary location. They are enclosed within bracket-equal delimiters, `[=` and `=]`. An equivalence class generally is designed to deal with primary-secondary sorting; that is, for languages like French that define groups of characters as sorting to the same primary location, and then have a tie-breaking, secondary sort. For example, if `e`, `é`, and `ê` belong to the same equivalence class, then `[[=e=]fg]`, `[[=é=]fg]`, and `[[=ê=]fg]` are each equivalent to `[eéêfg]`. For more information on collating sequences and their use, see the discussion on using internationalization features in the *Command and Shell User's Guide*.

1.2 Using the grep Command

The name of the `grep` command is an acronym for "global regular expression printer." The `egrep` and `fgrep` commands, allied to `grep`, are obsolescent and should be replaced with `grep -E` and `grep -F`, respectively. The differences in the way `grep` behaves when used with these flags are summarized in Table 1-3.

Table 1-3: Behavior of the grep Command

grep Version	Description
`grep`	"Basic `grep`" patterns (for `grep` with neither the `-E` nor the `-F` flag) are interpreted as basic regular expressions.
`grep -E` (`egrep`)	"Extended `grep`" patterns are interpreted as extended regular expressions.
`grep -F` (`fgrep`)	"Fixed `grep`" patterns are fixed strings; all regular expression operators are interpreted literally.

All forms of the `grep` command allow you to specify more than one expression as a multiline list. Surround the list with apostrophes, and separate the expressions with newline characters, as in this example using the Bourne shell:

```
$ strings hpcalc | grep -F 'math.h
> fatal.h'
```

In the C shell, you must enter a backslash before each newline character:

```
% strings hpcalc | grep -F 'math.h\
fatal.h'
```

You can also use the `-e` flag to specify multiple expressions on one line. For example:

```
% grep -e 'ab*c' -e 'de*f' myfile
```

By default, the `grep` command finds each line containing a match for the expression or expressions you specify. Table 1-4 describes command-line flags that allow you to specify other results from your searches.

Table 1-4: Flags for the grep Command

Flag	Description
`-b`	Precedes each output line with its disk block number. This flag is of use primarily to programmers who are trying to identify specific blocks on a disk by searching for the information contained in them.
`-c`	Counts matching lines and prints only the count.
`-e` *pattern_list*	Specifies matching on *pattern_list*; multiple patterns must be separated with newlines. Useful if *pattern_list* begins with a minus sign (-).
`-f` *pattern_file*	Uses the contents of *pattern_file* to supply the expressions to be matched. Specify one expression per line in *pattern_file*.
`-h`	Suppresses reporting of file names when multiple files are processed.
`-l`	Lists only the names of files containing matching lines. Each file name is listed only once, even if the file contains multiple matches. If standard input is specified among the files to be processed with this flag, `grep` returns the parenthesized phrase `(standard input)` for the file name on relevant matches.
`-n`	Precedes each matching line with its line number.
`-p` *paragraph_sep*	Uses *paragraph_sep* as a paragraph separator, and displays the entire paragraph containing each matched line. Does not display the paragraph separator lines. The default paragraph separator is a blank line.
`-q`	Operates in "quiet" mode, printing nothing except error messages. [Footnote 1]
`-s`	Suppresses error messages arising from nonexistent or unreadable files. Other error messages are still displayed. [Footnote 1]
`-v`	Outputs only lines that do not match the specified expressions.
`-w` *expr*	Matches only if *expr* is found as a separate word in the text. A word is any string of alphanumeric characters (letters, numbers, and underscores) delimited by nonalphanumeric characters (punctuation or white space) or by the beginning or end of the line. For example, `word1` is a word; `A+B` is not a word.

Flag	Description
`-x`	Outputs only lines matched in their entirety.
`-y`	Exact synonym for `-i`.

See the `grep`(1) reference page for more information about `grep` and regular expressions.
