**Regular Expressions Unfettered**

Regular expressions are a powerful mechanism for text processing. You can use regular expressions to search for a pattern within a block of text, to replace bits of that text with other bits of text, and to manipulate strings in various other subtle and interesting ways.

The shell itself does not support regular expressions natively. To use regular expressions, you must invoke an external tool.

Some tools that support regular expressions include:

* [awk](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/awk.1.html#//apple_ref/doc/man/1/awk)—A scripting language in and of itself. Described further in [“How AWK-ward.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/Howawk-ward/Howawk-ward.html#//apple_ref/doc/uid/TP40004268-TP40003518-SW10)
* [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep)—Returns the list of lines that match an expression (or the lines that do not match with the -v flag). Exits with a status of true (0) if a match occurred or false (1) if no match occurred.
* [perl](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/perl.1.html#//apple_ref/doc/man/1/perl)—A scripting language with more advanced regular expression functionality.
* [sed](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/sed.1.html#//apple_ref/doc/man/1/sed)—A tool that performs text substitutions based on regular expressions.

You will see these commands used throughout this chapter.

For the purposes of this chapter, you should paste the following lines of text into a text file with UNIX line endings (newline):

|  |
| --- |
| Mary had a little lamb, |
| its fleece was white as snow, |
| and everywhere that Mary went, |
| the lamb was sure to go. |
| A few more lines to confuse things: |
| Marylamb had a little. |
| This is a test. This is only a test. |
| Mary was married. A lamb was nearby. |
| Mary, a little lamb, and my grocer's freezer... |
| Mary a lamb. |
| Marry a lamb. |
| Mary had a lamb looked like a lamb. |
| I want chocolate for Valentine's day. |
| This line contains a slash (/). |
| This line contains a backslash (\). |
| This line contains brackets ([]). |
| Why is mary lowercase? |
| What about Mary, Mary, and Mary? |
| const people fox |
| constant turtles bear |
| constellation Libra |
| How about 9 \* 9? |
| The quick brown fox jumped over the lazy dog. |

Save this into a file called poem.txt.

**Where Can I Use Regular Expressions?**

Regular expressions are most commonly used for text filtering. For example, to change every occurrence of the letter "a" in a string to a capital "A", you might echo the string and pipe the result to sed like this:

|  |
| --- |
| echo "This is a test, this is only a test" | sed 's/a/A/g' |

You can also use regular expressions to search for strings in a file or a block of text by using the [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep) command. For example, to look for the word "bar" in the file foo.txt, you might do this:

|  |
| --- |
| grep "bar" foo.txt |
| # or |
| cat foo.txt | grep "bar" |

Finally, on occasion, it can be useful to use regular expressions in control statements. This advanced usage is described further in [“Using Regular Expressions in Control Statements.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW35)

**Types of Regular Expressions**

There are three basic types of regular expressions: basic regular expressions, extended regular expressions, and Perl regular expressions. Throughout this chapter, the sections points out areas in which they diverge. This section is just a summary of the differences. For more detail, see the appropriate section.

Basic regular expressions and extended regular expressions differ in the following areas:

* Basic regular expressions use a backslash prior to grouping/capturing parentheses (and prior to pipe operators within these parentheses). Extended regular expressions do not. These operators are described in [“Grouping Operators.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW13)
* Basic regular expressions use a backslash prior to a plus sign when used to mean “one or more of the previous character or group”. Extended regular expressions do not. This operator is described in [“Wildcards and Repetition Operators.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW3)
* Basic regular expressions use a backslash prior to a question mark when used to mean “zero or one of the previous character or group”. Extended regular expressions do not. This operator is described in [“Wildcards and Repetition Operators.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW3)

Perl regular expressions are equivalent to extended regular expressions with a few additional features:

* Perl can (optionally) use a dollar sign instead of a backslash to represent variables in substitution patterns, as described in [“Capturing Operators and Variables.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1)
* Perl supports noncapturing parentheses, as described in [“Noncapturing Parentheses.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW8)
* The order of multiple options within parentheses can be important when substrings come into play, as described in [“Grouping Operators.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW13)
* Perl allows you to include a literal square bracket anywhere within a character class by preceding it with a backslash, as described in [“Quoting Special Characters.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW15)
* Perl adds a number of additional switches that are equivalent to certain special characters and character classes. These are described in [“Character Class Shortcuts.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW5)
* Perl supports a broader range of modifiers. These are described in [“Using Modifiers.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW17)

**Regular Expression Syntax**

The fundamental format for regular expressions is one of the following, depending on what you are trying to do:

|  |
| --- |
| /search\_pattern/modifiers |
| command/search\_pattern/modifiers |
| command/search\_pattern/replacement/modifiers |

The first syntax is a basic search syntax. In the absence of a command prefix, such a regular expression returns the lines matching the search pattern. In some cases, the slash marks may be (or must be) omitted—in the pattern argument to the [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep) command, for example.

The second syntax is used for most commands. In this form, some operation occurs on lines matching the pattern. This may be a form of matching, or it may involve removing the portions of the line that match the pattern.

The third syntax is used for substitution commands. These can be thought of as a more complex form of search and replace.

For example, the following command searches for the word 'test' within the specified file:

|  |
| --- |
| # Expression: /test/ |
| grep 'test' poem.txt |

**Note:** Note that grep expects the leading and trailing slashes in the regular expression to be removed.

The availability of commands and flags varies somewhat between regular expression variants, and is described in the relevant sections.

**Positional Anchors and Flags**

A common way to significantly alter regular expression matching is through the use of positional anchors and flags.

Positional anchors allow you to specify the position within a line of text where an expression is allowed to match. There are two positional anchors that are regularly used: caret (^) and dollar ($). When placed at the beginning or end of an expression, these match the beginning and end of a line of text, respectively.

For example:

|  |
| --- |
| # Expression: /^Mary/ |
| grep "^Mary" < poem.txt |

This matches the word "Mary", but only when it appears at the beginning of a line. Similarly, the following matches the word "fox," but only at the end of a line:

|  |
| --- |
| # Expression: /fox$/ |
| grep "fox$" < poem.txt |

The other common technique for altering the matching behavior of a regular expression is through the use of flags. These flags, when placed at the end of a regular expression, can change whether a regular expression is allowed to match across multiple lines, whether the matching is case sensitive or insensitive, and various other aspects of matching.

**Note:** Different tools support different flags, and not all flags are supported with all tools. The grep command-line tool uses command-line flags instead of flags in the expression itself.

The most commonly used flag is the global flag. By default, only the first occurrence of a search term is matched. This is mainly of concern when performing substitutions. The global flag changes this so that a substitution alters every match in the line instead of just the first one.

For example:

|  |
| --- |
| # Expression: s/Mary/Joe/ |
| sed "s/Mary/Joe/" < poem.txt |

This replaces only the first occurrence of "Mary" with "Joe." By adding the global flag to the expression, it instead replaces every occurrence, as shown in the following example:

|  |
| --- |
| # Expression s/Mary/Joe/g |
| sed "s/Mary/Joe/g" < poem.txt |

**Wildcards and Repetition Operators**

One of the most common ways to enhance searching through regular expressions is with the use of wildcard matching.

A wildcard is a symbol that takes the place of any other symbol. In regular expressions, a period (.) is considered a wildcard, as it matches any single character. For example:

|  |
| --- |
| # Expression: /wa./ |
| grep 'wa.' poem.txt |

This matches lines containing both "was" and "want" because the dot can match any character.

Wildcards are typically combined with repetition operators to match lines in which only a portion of the content is known. For example, you might want to search for every line containing "Mary" with the word "lamb" appearing later. You might specify the expression like this:

|  |
| --- |
| # Expression: /Mary.\*lamb/ |
| grep "Mary.\*lamb" poem.txt |

This searches for Mary followed by zero or more characters, followed by lamb.

Of course, you probably want at least one character between those to avoid matches for strings containing "Marylamb". The most common way to solve this is with the plus (+) operator. However, you can construct this expression in several ways:

|  |
| --- |
| # Expression (Basic): /Mary.\+lamb/ |
| # Expression (Extended): /Mary.+lamb/ |
| # Expression: /Mary..\*lamb/ |
| grep "Mary.\+lamb" poem.txt |
| grep -E "Mary.+lamb" poem.txt # extended regexp |
| grep "Mary..\*lamb" poem.txt |

**Note:** The appearance of the plus operator differs depending on whether you are using basic or extended regular expressions; in basic regular expressions, it must be preceded by a backslash.

The first dot in the third expression matches a single character. The dot-asterisk afterwards matches be zero or more additional characters. Thus, these three statements are equivalent.

The final useful repetition operator is the question mark operator (?). This operator matches zero or one repetitions of whatever precedes it.

**Note:** Like the plus operator, this differs in appearance depending on whether you are using basic or extended regular expressions; in basic regular expressions, it must be preceded by a backslash.

For example, if you want to match both Mary and Marry, you might use an expression like this:

|  |
| --- |
| # Expression (Basic): /Marr\?y/ |
| # Expression (Extended): /Marr?y/ |
| grep "Marr\?y" poem.txt |
| grep -E "Marr?y" poem.txt |

The question mark causes the preceding r to be optional, and thus, this expression matches lines containing either “Mary” or “Marry.”

In summary, the basic wildcard and repetition operators are:

* period (.)—wildcard; matches a single character.
* question mark (\? or ?)—matches 0 or 1 of the previous character, grouping, or wildcard. (This operator differs depending on whether you are using basic or extended regular expressions.)
* asterisk(\*)—matches zero or more of the previous character, grouping, or wildcard.
* plus(\+ or +)—matches one or more of the previous character, grouping, or wildcard. (This operator differs depending on whether you are using basic or extended regular expressions.)

**Character Classes and Groups**

Searching for certain keywords can be useful, but it is often not enough. It is often useful to search for the presence or absence of key characters at a given position in a search string.

For example, assume that you require the words Mary and lamb to be within the same sentence. To do this, you need to only allow certain characters to appear between the two words. This can be achieved through the use of character classes.

There are two basic types of character classes: predefined character classes and custom, or user-defined character classes. These are described in the following sections.

**Predefined Character Classes**

Most regular expression languages support some form of predefined character classes. When used between brackets, these define commonly used sets of characters. The most broadly supported set of predefined character classes are the POSIX character classes:

* [:alnum:]—all alphanumeric characters (a-z, A-Z, and 0-9).
* [:alpha:]—all alphabetic characters (a-z, A-Z).
* [:blank:]—all whitespace within a line (spaces or tabs).
* [:cntrl:]—all control characters (ASCII 0-31).
* [:digit:]—all numbers.
* [:graph:]—all alphanumeric or punctuation characters.
* [:lower:]—all lowercase letters (a-z).
* [:print:]—all printable characters (opposite of [:cntrl:], same as the union of [:graph:] and [:space:]).
* [:punct:]—all punctuation characters
* [:space:]—all whitespace characters (space, tab, newline, carriage return, form feed, and vertical tab). (See note below about compatibility.)
* [:upper:]—all uppercase letters.
* [:xdigit:]—all hexadecimal digits (0-9, a-f, A-F).

For example, the following is another way to match any sentence containing Mary and lamb (but not if there are punctuation marks between them):

|  |
| --- |
| # Expression: /Mary[[:alpha:][:digit:][:blank:]][[:alpha:][:digit:][:blank:]]\*lamb/ |
| grep 'Mary[[:alpha:][:digit:][:blank:]][[:alpha:][:digit:][:blank:]]\*lamb' poem.txt |

**Compatibility Note:** Not all tools fully support POSIX character classes. In particular:

* The [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep) tool does not support [:space:] because this character class includes line break characters, which makes no sense in a tool that is designed to print lines that match a pattern.
* The [sed](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/sed.1.html#//apple_ref/doc/man/1/sed) tool accepts [:space:] but treats it like [:blank:] for the same reason.

**Custom Character Classes**

In addition to the predefined character classes, regular expression languages also allow custom, user-defined character classes. These custom character classes just look like a list of characters surrounded by square brackets.

For example, if you only want to allow spaces and letters, you might create a character class like this one:

|  |
| --- |
| # Expression: /Mary[a-z A-Z]\*lamb/ |
| grep "Mary[a-z A-Z]\*lamb" poem.txt |

In this example, there are two ranges (‘a’ through ‘z’ and ‘A’ through ‘Z’) allowed, as well as the space character. Thus, any letter or space matches this pattern, but other things (including the period character) do not. Thus, this line matches the first line of the poem, but does not match the later line that begins with "Mary was married."

However, this pattern also did not match the line containing a comma, which was not really the intent. Listing every reasonable range of characters with a single omission would be prohibitively large, particularly if you want to include high ASCII characters, control characters, and other potentially unprintable characters.

Fortunately, there is another special operator, the caret (^). When placed as the first character of a character class, matching is reversed. Thus, the following expression matches any character other than a period:

|  |
| --- |
| # /Mary[^.]\*lamb/ |
| grep "Mary[^.]\*lamb" poem.txt |

**Grouping Operators**

As mentioned previously, regular expressions also have a notion of grouping. The purpose of grouping is to treat multiple characters as a single entity, usually for the purposes of modifying that entity with a repeat operator. This grouping is done using parentheses or quoted parentheses, depending on the regular expression dialect being used.

**Note:** The syntax for grouping also results in a capture. This process is described in [“Capturing Operators and Variables.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1)

For example, say that you want to search for any string that contains the word “Mary” followed optionally by the word “had", followed by the word “a”. You might write this expression like this:

|  |
| --- |
| #Expression (Basic): /Mary \(had \)\?a/ |
| #Expression (Extended): /Mary (had )?a/ |
| grep "Mary \(had \)\?a" poem.txt |
| grep -E "Mary (had )?a" poem.txt |

**Note:** The grouping operator and optional operator differ depending on which program is processing the regular expression. The tools [sed](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/sed.1.html#//apple_ref/doc/man/1/sed), [awk](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/awk.1.html#//apple_ref/doc/man/1/awk), and [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep) use basic regular expressions (by default), and thus, these operators must be quoted. Any tools that use extended regular expressions use the bare operators.

Also note that the -E flag enables extended regular expressions in grep.

The flag to enable extended regular expressions in sed differs among different versions of the tool. For this reason, you should use basic regular expressions if at all possible when working with sed.

You can also use the grouping syntax to provide multiple options, any one of which is treated as a match. Expressions enclosed in parentheses match any one of a series of smaller expressions separated by a pipe (|) operator. For example, to search for Mary, lamb, or had, you might use this expression:

|  |
| --- |
| #Expression (Basic): /\(Mary\|had\|lamb\)/ |
| #Expression (Extended): /(Mary|had|lamb)/ |
| grep '\(Mary\|had\|lamb\)' poem.txt |
| grep -E '(Mary|had|lamb)' poem.txt |

Because regular expressions generally match from left to right, you should be careful when working with multiple options that are substrings of one another during substitution and be sure to place the larger of the possible matches first. Some regular expression engines always take the longer match, while other regular expression engines always take the leftmost match.

For example, the following lines give the same result:

|  |
| --- |
| sed -E 's/(lamb|lamb,)/orange/' poem.txt |
| sed -E 's/(lamb,|lamb)/orange/' poem.txt |

However the following lines do not:

|  |
| --- |
| perl -pi.bak -e 's/(lamb|lamb,)/orange/' < poem.txt |
| perl -pi.bak -e 's/(lamb,|lamb)/orange/' < poem.txt |

In Perl, when the input contains the word “lamb” followed by a comma, the regular expression engine matches the word “lamb” first because it is the leftmost option. It replaces it with the word “orange” and leaves the comma. In the second option, because the version with a comma matches first, the comma is deleted if it is there.

You can, of course, also avoid this problem by writing the expression as:

|  |
| --- |
| perl -pi.bak -e 's/lamb,?/orange/' < poem.txt |

**Using Empty Subexpressions**

Sometimes, when working with groups, you may find it necessary to include an optional group. It may be tempting to write such an expression like this:

|  |
| --- |
| # Expression (Extended): /const(ant|ellation|) (.\*)/ |

In an odd quirk, however, some command-line tools do not appreciate an empty subexpression. There are two ways to solve this.

The easiest way is to make the entire group optional like this:

|  |
| --- |
| # Expression (Extended): /const(ant|ellation)? (.\*)/ |
| grep -E 'const(ant|ellation)? (.\*)' |

Alternately, an empty expression may be inserted after the vertical bar.

|  |
| --- |
| # Expression (Extended): /const(ant|ellation|()) (.\*)/ |
| grep -E "const(ant|ellation|()) (.\*)" poem.txt |

**Note:** If you are mixing capturing with grouping, this method creates an empty capture, which ends up in the buffer following the capture buffer for this group (more on this in [“Capturing Operators and Variables”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1)).

**Quoting Special Characters**

As seen in previous sections, a number of characters have special meaning in regular expressions. For example, character classes are surrounded by square brackets, and the dash and caret characters have special meaning. You might ask how you can search for one of these characters. This is where quoting comes in.

In regular expressions, certain nonletter characters may have some special meaning, depending on context. To treat these characters as an ordinary character, you can prefix them with a backslash character (\). This also means that the backslash character is special in any context, so to match a literal backslash character, you must quote it with a second backslash.

There is one exception, however. To make a close bracket be a member of a character class, you do not quote it. Instead, you make it be the first character in the class.

**Note:** Perl rules for extended regular expressions allow you to quote a close bracket anywhere within a character class. Perl also recognizes the syntax shown here, however.

For example, to search for any string containing a backslash or a close bracket, you might use the following regular expression:

|  |
| --- |
| # Expression: /[]\\]/ |
| grep '[]\\]' poem.txt |

It looks a bit cryptic, but it is really relatively straightforward. The outer slashes delimit the regular expression. The brackets immediately inside the outer slashes are character class delimiters. The first close bracket immediately follows the open bracket, which makes it match an actual close bracket character instead of ending the character class. The two backslashes afterwards are, in fact, a quoted backslash, which makes this character class match the literal backslash character.

As a general rule, at least in extended regular expressions, any nonalphanumeric character can safely be quoted whether it is necessary to do so or not. If quoting it is not necessary, the extra backslash is simply ignored. However, it is not always safe to quote letters or numbers, as these have special meanings in certain regular expression dialects, as described in [“Capturing Operators and Variables”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1) and [“Perl and Python Extensions.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW2) In addition, quoting parentheses may not do what you might expect in some dialects, as described in [“Capturing Operators and Variables.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1)

In basic regular expressions the behavior when quoting characters other than parentheses, curly braces, numbers, and characters within a character class is undefined.

**Capturing Operators and Variables**

In [“Wildcards and Repetition Operators,”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW3) this chapter described ways to create more complicated patterns to match for the search portion of a search and replace operation. This section describes more powerful operations for the replacement portion of a search and replace operation.

Capturing operators and variables are used to take pieces of the original input text, capture them while searching, and then substitute those bits into the middle of the replacement text.

The easiest way to explain capturing operators and variables is by example. Suppose you want to swap the words quick and lazy in the string, "The quick brown fox jumped over the lazy dog." You might write an expression like this:

|  |
| --- |
| # Expression (Basic): s/The \(.\*\) brown \(.\*\) the \(.\*\) dog/The \3 brown \2 the \1 dog/ |
| # Expression (Extended): s/The (.\*) brown (.\*) the (.\*) dog/The \3 brown \2 the \1 dog/ |

When you pass these expressions to [sed](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/sed.1.html#//apple_ref/doc/man/1/sed), the last line of poem.txt should become "The lazy brown fox jumped over the quick dog."

|  |
| --- |
| # Expression (Basic): s/The (.\*) brown (.\*) the (.\*) dog/The \3 brown \2 the \1 dog/ |
| sed "s/The \(.\*\) brown \(.\*\) the \(.\*\) dog/The \3 brown \2 the \1 dog/" < poem.txt |
|  |
| # Expression (Extended): s/The \(.\*\) brown \(.\*\) the \(.\*\) dog/The \3 brown \2 the \1 dog/ |
| sed -E "s/The (.\*) brown (.\*) the (.\*) dog/The \3 brown \2 the \1 dog/" < poem.txt |
|  |
| # Perl supports extended form, but also supports |
| # using a dollar sign for the variable name. (Note |
| # the use of single quotes to prevent the shell from |
| # doing variable substitution on $1, $2, and $3.) |
| perl -pi.bak -e "s/The (.\*) brown (.\*) the (.\*) dog/The \3 brown \2 the \1 dog/" < poem.txt |
| perl -pi.bak -e 's/The (.\*) brown (.\*) the (.\*) dog/The $3 brown $2 the $1 dog/' < poem.txt |

**Note:** The syntax of the capturing operator differs depending on whether you are using basic, extended, or Perl regular expressions.

**Compatibility Note:** The use of the -E flag with sed to enable extended regular expressions varies from one operating system to another. For maximum portability, you should avoid using extended regular expressions with sed.

The content between each pair of parentheses (in this case—see note) is captured into its own buffer, numbered consecutively. Thus, in this expression, the content between “the” and “brown” is captured into a buffer. Then, the content between “brown” and “the” is captured. Finally, the content between “the” and “dog” is captured.

In the replacement string, the delimiter words (“The”, “brown”, “the”, and “dog”) are inserted, and the contents of the capture buffers are inserted in the opposite order.

**Note:** By default, repetition operators (except the question mark operator) are greedy. By default, they match the longest possible string that matches the expression as a whole. For example:

|  |
| --- |
| # s/Mary.\*lamb/Joe/ |
| sed "s/Mary.\*lamb/Joe/" < poem.txt |

In the poem, the line “Mary had a lamb looked like a lamb.” becomes simply “Joe.”.

If you want to only match up to the *first* occurrence of “lamb”, you must either use a Perl regular expression dialect extension, as described in [“Nongreedy Wildcard Matching”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW6) or use a greedy regular expression from the other end of the string to replace the word “lamb” with another word that is known to not occur elsewhere in the input.

For example:

|  |
| --- |
| sed 's/lamb\(.\*\)$/UNMATCHABLE\1/' < poem.txt | sed 's/^.\*UNMATCHABLE/Joe/' |

This statement produces the line “Joe looked like a lamb.”

**Mixing Capturing and Grouping Operators**

Since parentheses serve both as capturing and grouping operators, use of grouping may result in unexpected consequences when capturing text in the same expression. For example, the following expression will behave very differently depending on input:

|  |
| --- |
| # Expression /const(ant)? (.\*)/ |

The text you probably intended to capture is in the second buffer, not the first.

**Note:** In the Perl version of extended regular expressions (as described in [“Noncapturing Parentheses”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW8)), you can use noncapturing parentheses to prevent the capture of the first portion, as show below:

|  |
| --- |
| /const(?:ant)? (.\*)/ |

However, if you are using most command-line tools, this extended syntax is not supported.

**Using Modifiers**

The overall behavior of a regular expression can be tuned using a number of modifiers. For example:

|  |
| --- |
| /foo/i |

In this example, the /i modifier makes the regular expression match in a case-insensitive fashion. Thus, this matches both “Foo” and “fOo”.

Not all commands and languages support all modifiers. For example, most versions of the [sed](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/sed.1.html#//apple_ref/doc/man/1/sed) command support only the /g modifier.

The basic modifiers are:

* /g—replace globally. Without this flag, a substitution command replaces only the first matching occurrence per line. With this flag, a substitution command also replaces subsequent matches.
* /i—use case insensitive matching (Perl extension; equivalent to grep -i).
* /m—multiline matching (Perl extension). the $ and ^ anchors should match at newline boundaries in addition to matching at the beginning an end of the string as a whole. The dot (.) does not match newline characters.
* /o—compile once (Perl extension). In Perl, if a regular expression includes a variable as part of the pattern, the regular expression engine must recompile the expression every time it is used because the variable contents might have changed.

If you know that the contents will not change after they are set the first time, the /o flag disables recompilation of the expression. For regular expressions that do not contain variables, this switch has no effect.

* /s—single-line matching (Perl extension). The $ and ^ anchors should *not* match at newline boundaries. With this modifier, they only match at the very beginning and end of the string as a whole. The dot (.) matches newline characters just like any other character.
* /x—extend readability (Perl extension). This mode causes matching to ignore all whitespace between tokens in the expression unless quoted or wrapped in brackets (in most languages) and to treat a hash mark (#) as the start of a single-line comment.

**Note:** Not all whitespace is ignored; multicharacter tokens like \d must not be split or they will be interpreted differently.

The purpose of this mode is to allow you to split complex regular expressions into multiple lines. For example, in Perl, you might detect a date like this:

|  |
| --- |
| if ($foo =~ /(\d\d\d\d) # year |
| \s\*-\s\* # separator |
| (\d\d) # month |
| \s\*-\s\* # separator |
| (\d\d) # day |
| /x) { |
| print "Date detected\n"; |
| } |

The syntactical details vary from language to language.

**Perl and Python Extensions**

The regular expression dialect used in Perl, Python, and many other languages, are a further extension of extended regular expressions. Some of the major differences include:

* Addition of shortcuts for character classes. See [“Character Class Shortcuts.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW5)
* Addition of quotation operators. In a regular expression, the contents of variables appearing between \Q and \E are automatically quoted, and thus treated as literal text even if the variable contains characters that ordinarily have special meaning in a regular expression. These operators are useful when user input, stored in a Perl variable, is used as part of a regular expression.
* Support for retrieving captured values outside the scope of the expression; the captured values are stored in the variables $1, $2, and so on. (See [“Capturing Operators and Variables”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW1) for information about capturing parts of a regular expression.)

**Note:** In PHP, these captured values are passed back in an array that you can provide as an optional argument.

* Addition of nongreedy matching. See [“Nongreedy Wildcard Matching”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW6) for more information.
* Noncapturing parentheses. See [“Noncapturing Parentheses”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW8) for more information.

You can find links to additional resources that describe these extensions in [“For More Information.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW34)

**Character Class Shortcuts**

Perl regular expressions add a number of additional character class shortcuts. Some of these are listed below:

* \A—anchors matching to the beginning of the string as a whole (but not the beginning of lines within the string).

This shortcut is not broadly supported outside of Perl. In other languages, use ^ and add the /s modifier (or do not specify the /m modifier, depending) to specify line-at-once matching.

* \b—word boundary (see note).
* \B—nonword boundary (see note).
* \d—equivalent to [:digit:].
* \D—equivalent to [^:digit:].
* \f—form feed.
* \n—newline.
* \p—character matching a Unicode character property that follows. For example, \p{L} matches a Unicode letter.
* \P—character not matching a Unicode property that follows. For example, \P{L} matches any Unicode character that is not a letter.
* \r—carriage return.
* \s—equivalent to [:space:].
* \S—equivalent to [^:space:].
* \t—tab.
* \u—a single Unicode character in JavaScript regular expressions. This shortcut must be followed by four hexadecimal digits.
* \v—vertical tab.
* \w—equivalent to [:word:].
* \W—equivalent to [^:word:].
* \x—start of an ASCII character code (in hex). For example, \x20 is a space.
* \X—a single Unicode character (not supported universally). This shortcut must be followed by four hexadecimal digits.
* \z—anchors matching to the end of the string as a whole (but not the end of lines within the string).

This shortcut is not broadly supported outside of Perl. In other languages, use $ and add the /s modifier (or do not specify the /m modifier, depending) to specify line-at-once matching.

* \Z—anchors matching to the end of the string as a whole (but not the end of lines within the string). In some languages (including Perl), this matches prior to the closing line break if the string ends with a line break. To avoid this, use \z instead.

This shortcut is not broadly supported outside of Perl. In other languages, use $ and add the /s modifier to specify line-at-once matching.

These can be used anywhere on the left side of a regular expression, including within character classes.

**Note:** Word boundaries (the \b and \B switches) do not exist in basic or non-Perl extended regular expressions. These match the position between two characters rather than an actual character.

A word boundary occurs before the first character of a line (if it is a word character), at the end of the line (if it ends in a word character), and between any word character and nonword character that occur consecutively.

For substitution purposes, “replacing” a word boundary with text is equivalent to inserting that text, much like replacing other anchors such as ^ or $.

**Nongreedy Wildcard Matching**

By default, repeat operators are greedy, matching as many times as possible before attempting to match the next part of the string. This will generally result in the longest possible string that matches the expression as a whole. In some cases, you may want the matching to stop at the shortest possible string that matches the entire expression.

To support this, Perl regular expressions (along with many other dialects) supports nongreedy wildcard matching. To convert a greedy repeat operator to a nongreedy repeat operator, you just add a question mark after it.

For example, consider the nursery rhyme “Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went, the lamb was sure to go.” Assume that you apply the following expression:

|  |
| --- |
| /Mary.\*lamb/ |

That expression matches “Mary had a little lamb, its fleece was white as snow, and everywhere that Mary went, the lamb”.

Suppose that instead, you want to find the shortest possible string beginning with “Mary” and ending with “lamb”. You might instead use the following expression:

|  |
| --- |
| /Mary.\*?lamb/ |

That expression matches only the words “Mary had a little lamb”. The +? operator behaves similarly.

**Noncapturing Parentheses**

You may notice that the syntax for capture is identical to the syntax for grouping described in [“Wildcards and Repetition Operators.”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/RegularExpressionsUnfettered/RegularExpressionsUnfettered.html#//apple_ref/doc/uid/TP40004268-CH238-SW3) In most cases, the additional captures are not a problem. However, in some cases (particularly when splitting strings into arrays in Perl), you may wish to avoid capturing content if you are using parentheses merely as a grouping tool.

To turn off capturing for a given set of parentheses, add a question mark followed by a colon after the open parenthesis.

Consider the following example:

|  |
| --- |
| # Expression (Perl and Similar ONLY): /Mary (?:had)\* a little lamb\./ |
| perl -pi.bak -e "s/Mary (?:had )\*a little lamb\./Lovely day, isn't it?/" < poem.txt |

This expression matches “Mary”, followed by zero (0) or more instances of “had” followed by “a little lamb”, followed by a literal period, and replaces the offending line (“Mary had had a little lamb.”) with “Lovely day, isn't it?”.

**For More Information**

This chapter covers regular expressions as they apply to shell scripts. While it covers some of the more interesting extensions provided by languages such as Perl, it is by no means a complete reference to Perl regular expressions.

For a thorough explanation of Perl regular expressions and additional features and quirks in various programming languages, see <http://perldoc.perl.org/perlre.html> and <http://www.regular-expressions.info/>.

**Using Regular Expressions in Control Statements**

The shell’s [test](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/test.1.html#//apple_ref/doc/man/1/test) command (described in [“The test Command and Bracket Notation”](https://developer.apple.com/library/mac/documentation/opensource/conceptual/shellscripting/FlowControl,Expansion,andParsing/FlowControl,Expansion,andParsing.html#//apple_ref/doc/uid/TP40004268-CH4-SW5)) does not natively support regular expressions, so in order to use regular expressions in control statements, you must take advantage of the ability to execute arbitrary external commands (more specifically, the [grep](https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man1/grep.1.html#//apple_ref/doc/man/1/grep) command) instead of using bracket notation.

As shown throughout this chapter, the grep command takes a stream of text (or a path or list of paths) and prints every line that matches the specified regular expression. What you may not have noticed, however, is that its exit status changes depending on whether the input matches the specified expression.

The grep command exits with a successful exit status (0) if the input matches the specified expression at least once or a failed exit status (generally 1) if the pattern does not match. Thus, you can easily use it to control an if statement.

For example:

|  |
| --- |
| if (echo "$MYVAR" | grep "bar" > /dev/null) ; then |
| echo "The value of MYVAR ($MYVAR) contains \"bar\"." |
| fi |

In the above example, the rightmost exit status (from grep) is treated as the exit status for the group of commands (assuming that the echo command succeeds, which it always should). The redirect to /dev/null prevents the text output from being printed to the user’s screen.

**Performance Note:** Regular expressions should not be used if standard shell tests can do the same thing. Regular-expression-based tests are much slower than built-in shell tests because of the need to execute multiple external commands.

Regular expressions can also be used in other control statements such as while loops. For example, the following snippet counts the occurrences of the letter ‘x’ in a single-line string:

|  |
| --- |
| MYVAR="xxxxxx" |
| while (echo "$MYVAR" | grep 'x' > /dev/null) ; do |
| # Be sure to change MYVAR here! |
| echo "got x" |
| MYVAR="$(echo "$MYVAR" | sed -E 's/x//')" |
| done |

Of course, this contrived snippet is a good example of when you should avoid regular expressions; testing for an empty string makes this snippet run roughly twice as fast.