NAME

zshexpn − zsh expansion and substitution

DESCRIPTION

The following types of expansions are performed in the indicated order in five steps:

History Expansion
: This is performed only in interactive shells.

Alias Expansion
: Aliases are expanded immediately before the command line is parsed as explained under Aliasing in *zshmisc*(1).

Process Substitution
Parameter Expansion
Command Substitution
Arithmetic Expansion
Brace Expansion
: These five are performed in left−to−right fashion. On each argument, any of the five steps that are needed are performed one after the other. Hence, for example, all the parts of parameter expansion are completed before command substitution is started. After these expansions, all unquoted occurrences of the characters ‘\’,‘’’ and ‘"’ are removed.

Filename Expansion
: If the **SH_FILE_EXPANSION** option is set, the order of expansion is modified for compatibility with **sh** and **ksh**. In that case*filename e xpansion* is performed immediately after *alias expansion*, preceding the set of five expansions mentioned above.

Filename Generation
: This expansion, commonly referred to as **globbing**, is always done last.

The following sections explain the types of expansion in detail.

HISTORY EXPANSION

History expansion allows you to use words from previous command lines in the command line you are typing. This simplifies spelling corrections and the repetition of complicated commands or arguments.

Immediately before execution, each command is saved in the history list, the size of which is controlled by the **HISTSIZE** parameter. The one most recent command is always retained in any case. Each saved command in the history list is called a history *event* and is assigned a number, beginning with 1 (one) when the shell starts up. The history number that you may see in your prompt (see EXPANSION OF PROMPT SEQUENCES in *zshmisc*(1)) is the number that is to be assigned to the *next* command.

Overview

A history expansion begins with the first character of the **histchars** parameter, which is ‘**!**’ by default, and may occur anywhere on the command line, including inside double quotes (but not inside single quotes **’...’** or C−style quotes **$’...’** nor when escaped with a backslash).

The first character is followed by an optional event designator (see the section ‘Event Designators’) and then an optional word designator (the section ‘Word Designators’); if neither of these designators is present, no history expansion occurs.

Input lines containing history expansions are echoed after being expanded, but before any other expansions take place and before the command is executed. It is this expanded form that is recorded as the history event for later references.

History expansions do not nest.

By default, a history reference with no event designator refers to the same event as any preceding history reference on that command line; if it is the only history reference in a command, it refers to the previous command. However, if the option **CSH_JUNKIE_HISTORY** is set, then every history reference with no event specification *always* refers to the previous command.

For example, '**!**' is the event designator for the previous command, so '**!!:1**' always refers to the first word of the previous command, and '**!!$**' always refers to the last word of the previous command. With **CSH_JUNKIE_HISTORY** set, then '**!:1**' and '**!$**' function in the same manner as '**!!:1**' and '**!!$**', respectively. Conversely, if **CSH_JUNKIE_HISTORY** is unset, then '**!:1**' and '**!$**' refer to the first and last words, respectively, of the same event referenced by the nearest other history reference preceding them on the current command line, or to the previous command if there is no preceding reference.

The character sequence '*^foo^bar*' (where '**^**' is actually the second character of the **histchars** parameter) repeats the last command, replacing the string *foo* with *bar*. More precisely, the sequence '*^foo^bar^*' is synonymous with '**!!:s^***foo*^*bar*^', hence other modifiers (see the section 'Modifiers') may follow the final '**^**'. In particular, '*^foo^bar^***:G**' performs a global substitution.

If the shell encounters the character sequence '**!"**' in the input, the history mechanism is temporarily disabled until the current list (see *zshmisc*(1)) is fully parsed. The '**!"**' is removed from the input, and any subsequent '**!**' characters have no special significance.

A less convenient but more comprehensible form of command history support is provided by the **fc** builtin.

Event Designators

An event designator is a reference to a command−line entry in the history list. In the list below, remember that the initial **'!'** in each item may be changed to another character by setting the **histchars** parameter.

! Start a history expansion, except when followed by a blank, newline, '**=**' or '**(**'. If followed immediately by a word designator (see the section 'Word Designators'), this forms a history reference with no event designator (see the section 'Overview').

!! Refer to the previous command. By itself, this expansion repeats the previous command.

!*n* Refer to command−line *n*.

!−*n* Refer to the current command−line minus *n*.

!*str* Refer to the most recent command starting with *str*.

!?*str*[**?**] Refer to the most recent command containing *str*. The trailing '**?**' is necessary if this reference is to be followed by a modifier or followed by any text that is not to be considered part of *str*.

!# Refer to the current command line typed in so far. The line is treated as if it were complete up to and including the word before the one with the '**!#**' reference.

!{...**}** Insulate a history reference from adjacent characters (if necessary).

Word Designators

A word designator indicates which word or words of a given command line are to be included in a history reference. A '**:**' usually separates the event specification from the word designator. It may be omitted only if the word designator begins with a '**^**', '**$**', '*****', '**−**' or '**%**'. Word designators include:

0 The first input word (command).
n The *n*th argument.
^ The first argument. That is,**1**.
$ The last argument.
% The word matched by (the most recent) **?***str* search.
x−*y* A range of words; *x* defaults to **0**.
***** All the arguments, or a null value if there are none.
*x****** Abbreviates '*x*−**$**'.
x− Like '*x******' but omitting word **$**.

Note that a '**%**' word designator works only when used in one of '**!%**', '**!:%**' or '**!?***str***?:%**', and only when used after a **!?** expansion (possibly in an earlier command). Anything else results in an error, although the error may not be the most obvious one.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a '**:**'. These modifiers also work on the result of *filename generation* and *parameter*

expansion, except where noted.

a Turn a file name into an absolute path: prepends the current directory, if necessary; remove '**.**' path segments; and remove '**..**' path segments and the segments that immediately precede them.

This transformation is agnostic about what is in the filesystem, i.e. is on the logical, not the physical directory. It takes place in the same manner as when changing directories when neither of the options **CHASE_DOTS** or **CHASE_LINKS** is set. For example, '**/before/here/../after**' is always transformed to '**/before/after**', regardless of whether '**/before/here**' exists or what kind of object (dir, file, symlink, etc.) it is.

A Turn a file name into an absolute path as the '**a**' modifier does, and *then* pass the result through the **realpath(3)** library function to resolve symbolic links.

Note: on systems that do not have a **realpath(3)** library function, symbolic links are not resolved, so on those systems '**a**' and '**A**' are equivalent.

Note: **foo:A** and **realpath(foo)** are different on some inputs. For **realpath(foo)** semantics, see the '**P**' modifier.

c Resolve a command name into an absolute path by searching the command path given by the **PATH** variable. This does not work for commands containing directory parts. Note also that this does not usually work as a glob qualifier unless a file of the same name is found in the current directory.

e Remove all but the part of the filename extension following the '**.**'; see the definition of the filename extension in the description of the **r** modifier below. Note that according to that definition the result will be empty if the string ends with a '**.**'.

h [*digits*]
Remove a trailing pathname component, shortening the path by one directory level: this is the 'head' of the pathname. This works like '**dirname**'. If the**h** is follo wed immediately (with no spaces or other separator) by any number of decimal digits, and the value of the resulting number is non−zero, that number of leading components is preserved instead of the final component being removed. In an absolute path the leading '/' is the first component, so, for example, if **var=/my/path/to/something**, then **${var:h3}** substitutes **/my/path**. Consecutive '/'s are treated the same as a single '/'. In parameter substitution, digits may only be used if the expression is in braces, so for example the short form substitution **$var:h2** is treated as **${var:h}2**, not as **${var:h2}**. No restriction applies to the use of digits in history substitution or globbing qualifiers. If more components are requested than are present, the entire path is substituted (so this does not trigger a 'failed modifier' error in history expansion).

l Convert the words to all lowercase.

p Print the new command but do not execute it. Only works with history expansion.

P Turn a file name into an absolute path, like **realpath(3)**. The resulting path will be absolute, have neither '**.**' nor '**..**' components, and refer to the same directory entry as the input filename.

Unlike **realpath(3)**, non−existent trailing components are permitted and preserved.

q Quote the substituted words, escaping further substitutions. Works with history expansion and parameter expansion, though for parameters it is only useful if the resulting text is to be re−evaluated such as by **eval**.

Q Remove one level of quotes from the substituted words.

r Remove a filename extension leaving the root name. Strings with no filename extension are not altered. A filename extension is a '**.**' followed by any number of characters (including zero) that are neither '**.**' nor '/' and that continue to the end of the string. For example, the extension of '**foo.orig.c**' is '**.c**', and '**dir.c/foo**' has no extension.

s/*l*/*r*[/] Substitute *r* for *l* as described below. The substitution is done only for the first string that matches *l*. For arrays and for filename generation, this applies to each word of the expanded text. See

below for further notes on substitutions.

The forms '**gs**/*l*/*r*' and '**s**/*l*/*r*/**:G**' perform global substitution, i.e. substitute every occurrence of *r* for *l*. Note that the**g** or **:G** must appear in e xactly the position shown.

See further notes on this form of substitution below.

& Repeat the previous **s** substitution. Like **s**, may be preceded immediately by a **g**. In parameter expansion the **&** must appear inside braces, and in filename generation it must be quoted with a backslash.

t [*digits*]
Remove all leading pathname components, leaving the final component (tail). This works like '**basename**'. Any trailing slashes are first removed. Decimal digits are handled as described above for (h), but in this case that number of trailing components is preserved instead of the default 1; 0 is treated the same as 1.

u Convert the words to all uppercase.

x Like **q**, but break into words at whitespace. Does not work with parameter expansion.

The **s**/*l*/*r*/ substitution works as follows. By default the left−hand side of substitutions are not patterns, but character strings. Any character can be used as the delimiter in place of '/'. A backslash quotes the delimiter character. The character '**&**', in the right−hand−side *r*, is replaced by the text from the left−hand−side *l*. The '**&**' can be quoted with a backslash. A null *l* uses the previous string either from the previous *l* or from the contextual scan string *s* from '**!?***s*'. You can omit the rightmost delimiter if a newline immediately follows *r*; the rightmost '**?**' in a context scan can similarly be omitted. Note the same record of the last *l* and *r* is maintained across all forms of expansion.

Note that if a '**&**' is used within glob qualifiers an extra backslash is needed as a **&** is a special character in this case.

Also note that the order of expansions affects the interpretation of *l* and *r*. When used in a history expansion, which occurs before any other expansions, *l* and *r* are treated as literal strings (except as explained for **HIST_SUBST_PATTERN** below). When used in parameter expansion, the replacement of *r* into the parameter's value is done first, and then any additional process, parameter, command, arithmetic, or brace references are applied, which may evaluate those substitutions and expansions more than once if *l* appears more than once in the starting value. When used in a glob qualifier, any substitutions or expansions are performed once at the time the qualifier is parsed, even before the '**:s**' expression itself is divided into *l* and *r* sides.

If the option **HIST_SUBST_PATTERN** is set, *l* is treated as a pattern of the usual form described in the section FILENAME GENERATION below. This can be used in all the places where modifiers are available; note, however, that in globbing qualifiers parameter substitution has already taken place, so parameters in the replacement string should be quoted to ensure they are replaced at the correct time. Note also that complicated patterns used in globbing qualifiers may need the extended glob qualifier notation **(#q:s**/.../.../**)** in order for the shell to recognize the expression as a glob qualifier. Further, note that bad patterns in the substitution are not subject to the **NO_BAD_PATTERN** option so will cause an error.

When **HIST_SUBST_PATTERN** is set, *l* may start with a **#** to indicate that the pattern must match at the start of the string to be substituted, and a **%** may appear at the start or after an **#** to indicate that the pattern must match at the end of the string to be substituted. The **%** or **#** may be quoted with two backslashes.

For example, the following piece of filename generation code with the **EXTENDED_GLOB** option:

print −r −− *.c(#q:s/#%(#b)s(*).c/'S${match[1]}.C'/)

takes the expansion of ***.c** and applies the glob qualifiers in the **(#q**...**)** expression, which consists of a substitution modifier anchored to the start and end of each word (**#%**). This turns on backreferences (**(#b)**), so that the parenthesised subexpression is available in the replacement string as **${match[1]}**. The replacement string is quoted so that the parameter is not substituted before the start of filename generation.

The following **f**, **F**, **w** and **W** modifiers work only with parameter expansion and filename generation. They

are listed here to provide a single point of reference for all modifiers.

f Repeats the immediately (without a colon) following modifier until the resulting word doesn't change any more.

F:*expr***:** Like **f**, but repeats only *n* times if the expression *expr* evaluates to *n*. Any character can be used instead of the '**:**'; if '**(**', '**[**', or '**{**' is used as the opening delimiter, the closing delimiter should be '**)**', '**]**', or '**}**', respectively.

w Makes the immediately following modifier work on each word in the string.

W:*sep***:** Like **w** but words are considered to be the parts of the string that are separated by *sep*. Any character can be used instead of the '**:**'; opening parentheses are handled specially, see above.

PROCESS SUBSTITUTION

Each part of a command argument that takes the form '**<(***list***)**', '**>(***list***)**' or '**=(***list***)**' is subject to process substitution. The expression may be preceded or followed by other strings except that, to prevent clashes with commonly occurring strings and patterns, the last form must occur at the start of a command argument, and the forms are only expanded when first parsing command or assignment arguments. Process substitutions may be used following redirection operators; in this case, the substitution must appear with no trailing string.

Note that '**<<(***list***)**' is not a special syntax; it is equivalent to '**< <(***list***)**', redirecting standard input from the result of process substitution. Hence all the following documentation applies. The second form (with the space) is recommended for clarity.

In the case of the < or > forms, the shell runs the commands in *list* as a subprocess of the job executing the shell command line. If the system supports the **/dev/fd** mechanism, the command argument is the name of the device file corresponding to a file descriptor; otherwise, if the system supports named pipes (FIFOs), the command argument will be a named pipe. If the form with > is selected then writing on this special file will provide input for *list*. If< is used, then the file passed as an ar gument will be connected to the output of the *list* process. For example,

paste <(cut −f1 *file1***) <(cut −f3** *file2***) |**
tee >(*process1***) >(***process2***) >/dev/null**

cuts fields 1 and 3 from the files *file1* and *file2* respectively, pastes the results together, and sends it to the processes *process1* and *process2*.

If **=(...)** is used instead of **<(...)**, then the file passed as an argument will be the name of a temporary file containing the output of the *list* process. This may be used instead of the < form for a program that expects to lseek (see *lseek*(2)) on the input file.

There is an optimisation for substitutions of the form **=(<<<***arg***)**, where *arg* is a single−word argument to the here−string redirection <<<. This form produces a file name containing the value of *arg* after any substitutions have been performed. This is handled entirely within the current shell. This is effectively the reverse of the special form **$(<***arg***)** which treats *arg* as a file name and replaces it with the file's contents.

The = form is useful as both the **/dev/fd** and the named pipe implementation of **<(...)** have drawbacks. In the former case, some programmes may automatically close the file descriptor in question before examining the file on the command line, particularly if this is necessary for security reasons such as when the programme is running setuid. In the second case, if the programme does not actually open the file, the subshell attempting to read from or write to the pipe will (in a typical implementation, different operating systems may have different behaviour) block for ever and have to be killed explicitly. In both cases, the shell actually supplies the information using a pipe, so that programmes that expect to lseek (see *lseek*(2)) on the file will not work.

Also note that the previous example can be more compactly and efficiently written (provided the **MULTIOS** option is set) as:

paste <(cut −f1 *file1***) <(cut −f3** *file2***) **
> >(*process1***) > >(***process2***)**

The shell uses pipes instead of FIFOs to implement the latter two process substitutions in the above example.

There is an additional problem with **>(***process***)**; when this is attached to an external command, the parent shell does not wait for *process* to finish and hence an immediately following command cannot rely on the results being complete. The problem and solution are the same as described in the section *MULTIOS* in *zshmisc*(1). Hence in a simplified version of the example above:

paste <(cut −f1 *file1***) <(cut −f3** *file2***) > >(***process***)**

(note that no **MULTIOS** are involved), *process* will be run asynchronously as far as the parent shell is concerned. The workaround is:

{ paste <(cut −f1 *file1***) <(cut −f3** *file2***) } > >(***process***)**

The extra processes here are spawned from the parent shell which will wait for their completion.

Another problem arises any time a job with a substitution that requires a temporary file is disowned by the shell, including the case where '**&!**' or '**&|**' appears at the end of a command containing a substitution. In that case the temporary file will not be cleaned up as the shell no longer has any memory of the job. A workaround is to use a subshell, for example,

(mycmd =(myoutput)) &!

as the forked subshell will wait for the command to finish then remove the temporary file.

A general workaround to ensure a process substitution endures for an appropriate length of time is to pass it as a parameter to an anonymous shell function (a piece of shell code that is run immediately with function scope). For example, this code:

```
() {
  print File $1:
  cat $1
} =(print This be the verse)
```

outputs something resembling the following

```
File /tmp/zsh6nU0kS:
This be the verse
```

The temporary file created by the process substitution will be deleted when the function exits.

PARAMETER EXPANSION

The character '**$**' is used to introduce parameter expansions. See*zshpar am*(1) for a description of parameters, including arrays, associative arrays, and subscript notation to access individual array elements.

Note in particular the fact that words of unquoted parameters are not automatically split on whitespace unless the option **SH_WORD_SPLIT** is set; see references to this option below for more details. This is an important difference from other shells. However, as in other shells, null words are elided from unquoted parameters' expansions.

With default options, after the assignments:

```
array=("first word" "" "third word")
scalar="only word"
```

then **$array** substitutes two words, '**first word**' and '**third word**', and **$scalar** substitutes a single word '**only word**'. Note that second element of**array** w as elided. Scalar parameters can be elided too if their value is null (empty). To avoid elision, use quoting as follows: **"$scalar"** for scalars and **"${array[@]}"** or **"${(@)array}"** for arrays. (The last two forms are equivalent.)

Parameter expansions can involve *flags*, as in '**${(@kv)aliases}**', and other operators, such as '**${PREFIX:−"/usr/local"}**'. Parameter expansions can also be nested. These topics will be introduced below. The full rules are complicated and are noted at the end.

In the expansions discussed below that require a pattern, the form of the pattern is the same as that used for

filename generation; see the section 'Filename Generation'. Note that these patterns, along with the replacement text of any substitutions, are themselves subject to parameter expansion, command substitution, and arithmetic expansion. In addition to the following operations, the colon modifiers described in the section 'Modifiers' in the section 'History Expansion' can be applied: for example, **${i:s/foo/bar/}** performs string substitution on the expansion of parameter **$i**.

In the following descriptions, '*word*' refers to a single word substituted on the command line, not necessarily a space delimited word.

${*name***}**
: The value, if any, of the parameter *name* is substituted. The braces are required if the expansion is to be followed by a letter, digit, or underscore that is not to be interpreted as part of *name*. In addition, more complicated forms of substitution usually require the braces to be present; exceptions, which only apply if the option **KSH_ARRAYS** is not set, are a single subscript or any colon modifiers appearing after the name, or any of the characters '**^**', '**=**', '**~**', '**#**' or '**+**' appearing before the name, all of which work with or without braces.

  If *name* is an array parameter, and the **KSH_ARRAYS** option is not set, then the value of each element of *name* is substituted, one element per word. Otherwise, the expansion results in one word only; with **KSH_ARRAYS**, this is the first element of an array. No field splitting is done on the result unless the **SH_WORD_SPLIT** option is set. See also the flags **=** and **s:***string***:**.

${+*name***}**
: If *name* is the name of a set parameter '**1**' is substituted, otherwise '**0**' is substituted.

${*name***−***word***}**
${*name***:−***word***}**
: If *name* is set, or in the second form is non−null, then substitute its value; otherwise substitute *word*. In the second form*name* may be omitted, in which case *wor d* is always substituted.

${*name***+***word***}**
${*name***:+***word***}**
: If *name* is set, or in the second form is non−null, then substitute *word*; otherwise substitute nothing.

${*name***=***word***}**
${*name***:=***word***}**
${*name***::=***word***}**
: In the first form, if *name* is unset then set it to *word*; in the second form, if *name* is unset or null then set it to *word*; and in the third form, unconditionally set *name* to *word*. In all forms, the value of the parameter is then substituted.

${*name***?***word***}**
${*name***:?***word***}**
: In the first form, if *name* is set, or in the second form if *name* is both set and non−null, then substitute its value; otherwise, print *word* and exit from the shell. Interactive shells instead return to the prompt. If*wor d* is omitted, then a standard message is printed.

In any of the above expressions that test a variable and substitute an alternate *word*, note that you can use standard shell quoting in the *word* value to selectively override the splitting done by the **SH_WORD_SPLIT** option and the **=** flag, but not splitting by the **s:***string***:** flag.

In the following expressions, when *name* is an array and the substitution is not quoted, or if the '**(@)**' flag or the *name***[@]** syntax is used, matching and replacement is performed on each array element separately.

${*name***#***pattern***}**
${*name***##***pattern***}**
: If the *pattern* matches the beginning of the value of *name*, then substitute the value of *name* with the matched portion deleted; otherwise, just substitute the value of *name*. In the first form, the smallest matching pattern is preferred; in the second form, the largest matching pattern is preferred.

${*name***%***pattern***}**
${*name***%%***pattern***}**
If the *pattern* matches the end of the value of *name*, then substitute the value of *name* with the matched portion deleted; otherwise, just substitute the value of *name*. In the first form, the smallest matching pattern is preferred; in the second form, the largest matching pattern is preferred.

${*name***:#***pattern***}**
If the *pattern* matches the value of *name*, then substitute the empty string; otherwise, just substitute the value of *name*. If*name* is an array the matching array elements are remo ved (use the '**(M)**' flag to remove the non−matched elements).

${*name***:|***arrayname***}**
If *arrayname* is the name (N.B., not contents) of an array variable, then any elements contained in *arrayname* are removed from the substitution of *name*. If the substitution is scalar, either because *name* is a scalar variable or the expression is quoted, the elements of *arrayname* are instead tested against the entire expression.

${*name***:***arrayname***}**
Similar to the preceding substitution, but in the opposite sense, so that entries present in both the original substitution and as elements of *arrayname* are retained and others removed.

${*name***:^***arrayname***}**
${*name***:^^***arrayname***}**
Zips two arrays, such that the output array is twice as long as the shortest (longest for '**:^^**') of **name** and **arrayname**, with the elements alternatingly being picked from them. For '**:^**', if one of the input arrays is longer, the output will stop when the end of the shorter array is reached. Thus,

a=(1 2 3 4); b=(a b); print ${a:^b}

will output '**1 a 2 b**'. For '**:^^**', then the input is repeated until all of the longer array has been used up and the above will output '**1 a 2 b 3 a 4 b**'.

Either or both inputs may be a scalar, they will be treated as an array of length 1 with the scalar as the only element. If either array is empty, the other array is output with no extra elements inserted.

Currently the following code will output '**a b**' and '**1**' as two separate elements, which can be unexpected. The second print provides a workaround which should continue to work if this is changed.

a=(a b); b=(1 2); print −l "${a:^b}"; print −l "${${a:^b}}"

${*name***:***offset***}**
${*name***:***offset***:***length***}**
This syntax gives effects similar to parameter subscripting in the form **$***name***[***start***,***end***]**, but is compatible with other shells; note that both *offset* and *length* are interpreted differently from the components of a subscript.

If *offset* is non−negative, then if the variable *name* is a scalar substitute the contents starting *offset* characters from the first character of the string, and if *name* is an array substitute elements starting *offset* elements from the first element. If *length* is given, substitute that many characters or elements, otherwise the entire rest of the scalar or array.

A positive *offset* is always treated as the offset of a character or element in *name* from the first character or element of the array (this is different from native zsh subscript notation). Hence 0 refers to the first character or element regardless of the setting of the option **KSH_ARRAYS**.

A negative offset counts backwards from the end of the scalar or array, so that −1 corresponds to the last character or element, and so on.

When positive, *length* counts from the *offset* position toward the end of the scalar or array. When negative, *length* counts back from the end. If this results in a position smaller than *offset*, a diagnostic is printed and nothing is substituted.

The option **MULTIBYTE** is obeyed, i.e. the offset and length count multibyte characters where appropriate.

offset and *length* undergo the same set of shell substitutions as for scalar assignment; in addition, they are then subject to arithmetic evaluation. Hence, for example

print ${foo:3}
print ${foo: 1 + 2}
print ${foo:$((1 + 2))}
print ${foo:$(echo 1 + 2)}

all have the same effect, extracting the string starting at the fourth character of **$foo** if the substitution would otherwise return a scalar, or the array starting at the fourth element if **$foo** would return an array. Note that with the option **KSH_ARRAYS $foo** always returns a scalar (regardless of the use of the offset syntax) and a form such as **${foo[*]:3}** is required to extract elements of an array named **foo**.

If *offset* is negative, the − may not appear immediately after the **:** as this indicates the **${***name***:−***word***}** form of substitution. Instead, a space may be inserted before the −. Furthermore, neither *offset* nor *length* may begin with an alphabetic character or **&** as these are used to indicate history−style modifiers. To substitute a value from a variable, the recommended approach is to precede it with a **$** as this signifies the intention (parameter substitution can easily be rendered unreadable); however, as arithmetic substitution is performed, the expression **${var: offs}** does work, retrieving the offset from **$offs**.

For further compatibility with other shells there is a special case for array offset 0. This usually accesses the first element of the array. However, if the substitution refers to the positional parameter array, e.g. **$@** or **$***, then offset 0 instead refers to **$0**, offset 1 refers to **$1**, and so on. In other words, the positional parameter array is effectively extended by prepending **$0**. Hence**${*:0:1}** substitutes **$0** and **${*:1:1}** substitutes **$1**.

${*name***/***pattern***/***repl***}**
${*name***//***pattern***/***repl***}**
${*name***:/***pattern***/***repl***}**

Replace the longest possible match of *pattern* in the expansion of parameter *name* by string *repl*. The first form replaces just the first occurrence, the second form all occurrences, and the third form replaces only if *pattern* matches the entire string. Both *pattern* and *repl* are subject to double−quoted substitution, so that expressions like **${name/$opat/$npat}** will work, but obey the usual rule that pattern characters in **$opat** are not treated specially unless either the option **GLOB_SUBST** is set, or **$opat** is instead substituted as **${~opat}**.

The *pattern* may begin with a '**#**', in which case the *pattern* must match at the start of the string, or '**%**', in which case it must match at the end of the string, or '**#%**' in which case the *pattern* must match the entire string. The *repl* may be an empty string, in which case the final '/' may also be omitted. To quote the final '/' in other cases it should be preceded by a single backslash; this is not necessary if the '/' occurs inside a substituted parameter. Note also that the '**#**', '**%**' and '**#%** are not active if they occur inside a substituted parameter, even at the start.

If, after quoting rules apply, **${***name***}** expands to an array, the replacements act on each element individually. Note also the effect of the **I** and **S** parameter expansion flags below; however, the flags **M**, **R**, **B**, **E** and **N** are not useful.

For example,

foo="twinkle twinkle little star" sub="t*e" rep="spy"
print ${foo//${~sub}/$rep}
print ${(S)foo//${~sub}/$rep}

Here, the '**~**' ensures that the text of **$sub** is treated as a pattern rather than a plain string. In the first case, the longest match for **t*e** is substituted and the result is '**spy star**', while in the second case, the shortest matches are taken and the result is '**spy spy lispy star**'.

${#*spec***}**

If *spec* is one of the above substitutions, substitute the length in characters of the result instead of the result itself. If *spec* is an array expression, substitute the number of elements of the result. This has the side−effect that joining is skipped even in quoted forms, which may affect other sub−expressions in *spec*. Note that '^', '=', and '~', below, must appear to the left of '#' when these forms are combined.

If the option **POSIX_IDENTIFIERS** is not set, and *spec* is a simple name, then the braces are optional; this is true even for special parameters so e.g. **$#−** and **$#*** take the length of the string **$−** and the array **$*** respectively. If**POSIX_IDENTIFIERS** is set, then braces are required for the # to be treated in this fashion.

${^*spec***}**

Turn on the **RC_EXPAND_PARAM** option for the evaluation of *spec*; if the '^' is doubled, turn it off. When this option is set, array expansions of the form *foo***${**_xx_**}***bar*, where the parameter *xx* is set to **(***a b c***)**, are substituted with '*fooabar foobbar foocbar*' instead of the default '*fooa b cbar*'. Note that an empty array will therefore cause all arguments to be removed.

Internally, each such expansion is converted into the equivalent list for brace expansion. E.g., **${^var}** becomes **{$var[1],$var[2],...}**, and is processed as described in the section 'Brace Expansion' below: note, however, the expansion happens immediately, with any explicit brace expansion happening later. If word splitting is also in effect the **$var[***N***]** may themselves be split into different list elements.

${=*spec***}**

Perform word splitting using the rules for **SH_WORD_SPLIT** during the evaluation of *spec*, but regardless of whether the parameter appears in double quotes; if the '=' is doubled, turn it off. This forces parameter expansions to be split into separate words before substitution, using **IFS** as a delimiter. This is done by default in most other shells.

Note that splitting is applied to *word* in the assignment forms of *spec before* the assignment to *name* is performed. This affects the result of array assignments with the **A** flag.

${~*spec***}**

Turn on the **GLOB_SUBST** option for the evaluation of *spec*; if the '~' is doubled, turn it off. When this option is set, the string resulting from the expansion will be interpreted as a pattern anywhere that is possible, such as in filename expansion and filename generation and pattern−matching contexts like the right hand side of the '=' and '**!=**' operators in conditions.

In nested substitutions, note that the effect of the **~** applies to the result of the current level of substitution. A surrounding pattern operation on the result may cancel it. Hence, for example, if the parameter **foo** is set to ***, **${~foo//*/*.c}** is substituted by the pattern ***.c**, which may be expanded by filename generation, but **${${~foo}//*/*.c}** substitutes to the string ***.c**, which will not be further expanded.

If a **${...}** type parameter expression or a **$(...)** type command substitution is used in place of *name* above, it is expanded first and the result is used as if it were the value of *name*. Thus it is possible to perform nested operations: **${${foo#head}%tail}** substitutes the value of **$foo** with both '**head**' and '**tail**' deleted. The form with **$(...)** is often useful in combination with the flags described next; see the examples below. Each *name* or nested **${...}** in a parameter expansion may also be followed by a subscript expression as described in *Array Parameters* in *zshparam*(1).

Note that double quotes may appear around nested expressions, in which case only the part inside is treated as quoted; for example, **${(f)"$(foo)"}** quotes the result of **$(foo)**, but the flag '**(f)**' (see below) is applied using the rules for unquoted expansions. Note further that quotes are themselves nested in this context; for example, in **"${(@f)"$(foo)"}"**, there are two sets of quotes, one surrounding the whole expression, the other (redundant) surrounding the **$(foo)** as before.

Parameter Expansion Flags

If the opening brace is directly followed by an opening parenthesis, the string up to the matching closing parenthesis will be taken as a list of flags. In cases where repeating a flag is meaningful, the repetitions need not be consecutive; for example, '(**q%q%q**)' means the same thing as the more readable '(**%%qqq**)'. The following flags are supported:

Evaluate the resulting words as numeric expressions and output the characters corresponding to the resulting integer. Note that this form is entirely distinct from use of the**#** without parentheses.

If the **MULTIBYTE** option is set and the number is greater than 127 (i.e. not an ASCII character) it is treated as a Unicode character.

% Expand all **%** escapes in the resulting words in the same way as in prompts (see EXPANSION OF PROMPT SEQUENCES in *zshmisc*(1)). If this flag is given twice, full prompt expansion is done on the resulting words, depending on the setting of the **PROMPT_PERCENT**, **PROMPT_SUBST** and **PROMPT_BANG** options.

@ In double quotes, array elements are put into separate words. E.g., '**"${(@)foo}"**' is equivalent to '**"${foo[@]}"**' and '**"${(@)foo[1,2]}"**' is the same as '**"$foo[1]" "$foo[2]"**'. This is distinct from *field splitting* by the **f**, **s** or **z** flags, which still applies within each array element.

A Convert the substitution into an array expression, even if it otherwise would be scalar. This has lower precedence than subscripting, so one level of nested expansion is required in order that subscripts apply to array elements. Thus **${${(A)***name***}[1]}** yields the full value of *name* when *name* is scalar.

This assigns an array parameter with '**${**...**=**...**}**', '**${**...**:=**...**}**' or '**${**...**::=**...**}**'. If this flag is repeated (as in '**AA**'), assigns an associative array parameter. Assignment is made before sorting or padding; if field splitting is active, the *word* part is split before assignment. The *name* part may be a subscripted range for ordinary arrays; when assigning an associative array, the *word* part *must* be converted to an array, for example by using '**${(AA)=***name***=...}**' to activate field splitting.

Surrounding context such as additional nesting or use of the value in a scalar assignment may cause the array to be joined back into a single string again.

a Sort in array index order; when combined with '**O**' sort in reverse array index order. Note that '**a**' is therefore equivalent to the default but '**Oa**' is useful for obtaining an array's elements in reverse order.

b Quote with backslashes only characters that are special to pattern matching. This is useful when the contents of the variable are to be tested using **GLOB_SUBST**, including the **${~...}** switch.

Quoting using one of the **q** family of flags does not work for this purpose since quotes are not stripped from non−pattern characters by **GLOB_SUBST**. In other words,

```
pattern=${(q)str}
[[ $str = ${~pattern} ]]
```

works if **$str** is '**a*b**' but not if it is '**a b**', whereas

```
pattern=${(b)str}
[[ $str = ${~pattern} ]]
```

is always true for any possible value of **$str**.

c With **${#***name***}**, count the total number of characters in an array, as if the elements were concatenated with spaces between them. This is not a true join of the array, so other expressions used with this flag may have an effect on the elements of the array before it is counted.

C Capitalize the resulting words. 'Words' in this case refers to sequences of alphanumeric characters separated by non−alphanumerics, *not* to words that result from field splitting.

D Assume the string or array elements contain directories and attempt to substitute the leading part of these by names. The remainder of the path (the whole of it if the leading part was not substituted) is then quoted so that the whole string can be used as a shell argument. This is the reverse

of '**~**' substitution: see the section FILENAME EXPANSION below.

e Perform single word shell expansions, namely *parameter expansion*, *command substitution* and *arithmetic expansion*, on the result. Such expansions can be nested but too deep recursion may have unpredictable effects.

f Split the result of the expansion at newlines. This is a shorthand for '**ps:\n:**'.

F Join the words of arrays together using newline as a separator. This is a shorthand for '**pj:\n:**'.

g:*opts***:** Process escape sequences like the echo builtin when no options are given (**g::**). With the **o** option, octal escapes don't take a leading zero. With the **c** option, sequences like '**^X**' are also processed. With the **e** option, processes '**\M−t**' and similar sequences like the print builtin. With both of the **o** and **e** options, behaves like the print builtin except that in none of these modes is '**\c**' interpreted.

i Sort case−insensitively. May be combined with '**n**' or '**O**'.

k If *name* refers to an associative array, substitute the *keys* (element names) rather than the values of the elements. Used with subscripts (including ordinary arrays), force indices or keys to be substituted even if the subscript form refers to values. However, this flag may not be combined with subscript ranges. With the **KSH_ARRAYS** option a subscript '**[*]**' or '**[@]**' is needed to operate on the whole array, as usual.

L Convert all letters in the result to lower case.

n Sort decimal integers numerically; if the first differing characters of two test strings are not digits, sorting is lexical. Integers with more initial zeroes are sorted before those with fewer or none. Hence the array '**foo1 foo02 foo2 foo3 foo20 foo23**' is sorted into the order shown. May be combined with '**i**' or '**O**'.

o Sort the resulting words in ascending order; if this appears on its own the sorting is lexical and case−sensitive (unless the locale renders it case−insensitive). Sorting in ascending order is the default for other forms of sorting, so this is ignored if combined with '**a**', '**i**' or '**n**'.

O Sort the resulting words in descending order; '**O**' without '**a**', '**i**' or '**n**' sorts in reverse lexical order. May be combined with '**a**', '**i**' or '**n**' to reverse the order of sorting.

P This forces the value of the parameter *name* to be interpreted as a further parameter name, whose value will be used where appropriate. Note that flags set with one of the **typeset** family of commands (in particular case transformations) are not applied to the value of *name* used in this fashion.

If used with a nested parameter or command substitution, the result of that will be taken as a parameter name in the same way. For example, if you have '**foo=bar**' and '**bar=baz**', the strings **${(P)foo}**, **${(P)${foo}}**, and **${(P)$(echo bar)}** will be expanded to '**baz**'.

Likewise, if the reference is itself nested, the expression with the flag is treated as if it were directly replaced by the parameter name. It is an error if this nested substitution produces an array with more than one word. For example, if '**name=assoc**' where the parameter **assoc** is an associative array, then '**${${(P)name}[elt]}**' refers to the element of the associative subscripted '**elt**'.

q Quote characters that are special to the shell in the resulting words with backslashes; unprintable or invalid characters are quoted using the **$'***NNN***'** form, with separate quotes for each octet.

If this flag is given twice, the resulting words are quoted in single quotes and if it is given three times, the words are quoted in double quotes; in these forms no special handling of unprintable or invalid characters is attempted. If the flag is given four times, the words are quoted in single quotes preceded by a **$**. Note that in all three of these forms quoting is done unconditionally, even if this does not change the way the resulting string would be interpreted by the shell.

If a **q−** is given (only a single **q** may appear), a minimal form of single quoting is used that only quotes the string if needed to protect special characters. Typically this form gives the most readable output.

If a **q+** is given, an extended form of minimal quoting is used that causes unprintable characters to

be rendered using **$'**...**'**. This quoting is similar to that used by the output of values by the **typeset** family of commands.

Q Remove one level of quotes from the resulting words.

t Use a string describing the type of the parameter where the value of the parameter would usually appear. This string consists of keywords separated by hyphens ('**-**'). The first keyword in the string describes the main type, it can be one of '**scalar**', '**array**', '**integer**', '**float**' or '**association**'. The other keywords describe the type in more detail:

local for local parameters

left for left justified parameters

right_blanks
for right justified parameters with leading blanks

right_zeros
for right justified parameters with leading zeros

lower for parameters whose value is converted to all lower case when it is expanded

upper for parameters whose value is converted to all upper case when it is expanded

readonly
for readonly parameters

tag for tagged parameters

export for exported parameters

unique for arrays which keep only the first occurrence of duplicated values

hide for parameters with the 'hide' flag

hideval for parameters with the 'hideval' flag

special for special parameters defined by the shell

u Expand only the first occurrence of each unique word.

U Convert all letters in the result to upper case.

v Used with **k**, substitute (as two consecutive words) both the key and the value of each associative array element. Used with subscripts, force values to be substituted even if the subscript form refers to indices or keys.

V Make any special characters in the resulting words visible.

w With **${#***name***}**, count words in arrays or strings; the **s** flag may be used to set a word delimiter.

W Similar to **w** with the difference that empty words between repeated delimiters are also counted.

X With this flag, parsing errors occurring with the **Q**, **e** and **#** flags or the pattern matching forms such as '**${***name***#***pattern***}**' are reported. Without the flag, errors are silently ignored.

z Split the result of the expansion into words using shell parsing to find the words, i.e. taking into account any quoting in the value. Comments are not treated specially but as ordinary strings, similar to interactive shells with the **INTERACTIVE_COMMENTS** option unset (however, see the **Z** flag below for related options)

Note that this is done very late, even later than the '**(s)**' flag. So to access single words in the result use nested expansions as in '**${${(z)foo}[2]}**'. Likewise, to remove the quotes in the resulting words use '**${(Q)${(z)foo}}**'.

0 Split the result of the expansion on null bytes. This is a shorthand for '**ps:\0:**'.

The following flags (except **p**) are followed by one or more arguments as shown. Any character, or the matching pairs '**(...)**', '**{...}**', '**[...]**', or '<...>', may be used in place of a colon as delimiters, but note that when a flag takes more than one argument, a matched pair of delimiters must surround each argument.

p Recognize the same escape sequences as the **print** builtin in string arguments to any of the flags described below that follow this argument.

Alternatively, with this option string arguments may be in the form **$***var* in which case the value of the variable is substituted. Note this form is strict; the string argument does not undergo general parameter expansion.

For example,

```
sep=:
val=a:b:c
print ${(ps.$sep.)val}
```

splits the variable on a **:**.

~ Strings inserted into the expansion by any of the flags below are to be treated as patterns. This applies to the string arguments of flags that follow **~** within the same set of parentheses. Compare with **~** outside parentheses, which forces the entire substituted string to be treated as a pattern. Hence, for example,

```
[[ "?" = ${(~j.|.)array} ]]
```

treats '|' as a pattern and succeeds if and only if **$array** contains the string '**?**' as an element. The **~** may be repeated to toggle the behaviour; its effect only lasts to the end of the parenthesised group.

j:*string***:**
Join the words of arrays together using *string* as a separator. Note that this occurs before field splitting by the **s:***string***:** flag or the **SH_WORD_SPLIT** option.

l:*expr***::***string1***::***string2***:**
Pad the resulting words on the left. Each word will be truncated if required and placed in a field *expr* characters wide.

The arguments **:***string1***:** and **:***string2***:** are optional; neither, the first, or both may be given. Note that the same pairs of delimiters must be used for each of the three arguments. The space to the left will be filled with *string1* (concatenated as often as needed) or spaces if *string1* is not given. If both *string1* and *string2* are given, *string2* is inserted once directly to the left of each word, truncated if necessary, before *string1* is used to produce any remaining padding.

If either of *string1* or *string2* is present but empty, i.e. there are two delimiters together at that point, the first character of **$IFS** is used instead.

If the **MULTIBYTE** option is in effect, the flag **m** may also be given, in which case widths will be used for the calculation of padding; otherwise individual multibyte characters are treated as occupying one unit of width.

If the **MULTIBYTE** option is not in effect, each byte in the string is treated as occupying one unit of width.

Control characters are always assumed to be one unit wide; this allows the mechanism to be used for generating repetitions of control characters.

m Only useful together with one of the flags **l** or **r** or with the **#** length operator when the **MULTIBYTE** option is in effect. Use the character width reported by the system in calculating how much of the string it occupies or the overall length of the string. Most printable characters have a width of one unit, however certain Asian character sets and certain special effects use wider characters; combining characters have zero width. Non−printable characters are arbitrarily counted as zero width; how they would actually be displayed will vary.

If the **m** is repeated, the character either counts zero (if it has zero width), else one. For printable character strings this has the effect of counting the number of glyphs (visibly separate characters), except for the case where combining characters themselves have non−zero width (true in certain alphabets).

r:*expr***::***string1***::***string2***:**
As **l**, but pad the words on the right and insert *string2* immediately to the right of the string to be padded.

Left and right padding may be used together. In this case the strategy is to apply left padding to the first half width of each of the resulting words, and right padding to the second half. If the string to be padded has odd width the extra padding is applied on the left.

s:*string***:**
Force field splitting at the separator *string*. Note that a*string* of tw o or more characters means that all of them must match in sequence; this differs from the treatment of two or more characters in the **IFS** parameter. See also the = flag and the **SH_WORD_SPLIT** option. An empty string may also be given in which case every character will be a separate element.

For historical reasons, the usual behaviour that empty array elements are retained inside double quotes is disabled for arrays generated by splitting; hence the following:

```
line="one::three"
print -l "${(s.:.)line}"
```

produces two lines of output for **one** and **three** and elides the empty field. To override this behaviour, supply the '**(@)**' flag as well, i.e. **"${(@s.:.)line}"**.

Z:*opts***:** As **z** but takes a combination of option letters between a following pair of delimiter characters. With no options the effect is identical to **z**. **(Z+c+)** causes comments to be parsed as a string and retained; any field in the resulting array beginning with an unquoted comment character is a comment. **(Z+C+)** causes comments to be parsed and removed. The rule for comments is standard: anything between a word starting with the third character of **$HISTCHARS**, default #, up to the next newline is a comment. **(Z+n+)** causes unquoted ne wlines to be treated as ordinary whitespace, else they are treated as if they are shell code delimiters and converted to semicolons. Options are combined within the same set of delimiters, e.g. **(Z+Cn+)**.

:*flags***:** The underscore () flag is reserved for future use. As of this revision of zsh, there are no valid *flags*; anything following an underscore, other than an empty pair of delimiters, is treated as an error, and the flag itself has no effect.

The following flags are meaningful with the **${**...#...**}** or **${**...**%**...**}** forms. The **S** and **I** flags may also be used with the **${**.../.../**}** forms.

S With **#** or **##**, search for the match that starts closest to the start of the string (a 'substring match'). Of all matches at a particular position, # selects the shortest and ## the longest:

```
% str="aXbXc"
% echo ${(S)str#X*}
abXc
% echo ${(S)str##X*}
a
%
```

With **%** or **%%**, search for the match that starts closest to the end of the string:

```
% str="aXbXc"
% echo ${(S)str%X*}
aXbc
% echo ${(S)str%%X*}
aXb
%
```

(Note that **%** and **%%** don't search for the match that ends closest to the end of the string, as one might expect.)

With substitution via **${**.../.../**}** or **${**...//.../**}**, specifies non−greedy matching, i.e. that the shortest instead of the longest match should be replaced:

```
% str="abab"
% echo ${str/*b/_}
_
% echo ${(S)str/*b/_}
_ab
%
```

I:*expr***:** Search the *expr*th match (where *expr* evaluates to a number). This only applies when searching for substrings, either with the **S** flag, or with **${**.../...**}** (only the *expr*th match is substituted) or **${**...//...**}** (all matches from the *expr*th on are substituted). The default is to take the first match.

The *expr*th match is counted such that there is either one or zero matches from each starting position in the string, although for global substitution matches overlapping previous replacements are ignored. With the **${**...**%**...**}** and **${**...**%%**...**}** forms, the starting position for the match moves backwards from the end as the index increases, while with the other forms it moves forward from the start.

Hence with the string

which switch is the right switch for Ipswich?

substitutions of the form **${(SI:***N***:)string#w*ch}** as *N* increases from 1 will match and remove '**which**', '**witch**', '**witch**' and '**wich**'; the form using '**##**' will match and remove '**which switch is the right switch for Ipswich**', '**witch is the right switch for Ipswich**', '**witch for Ipswich**' and '**wich**'. The form using '**%**' will remove the same matches as for '**#**', but in reverse order, and the form using '**%%**' will remove the same matches as for '**##**' in reverse order.

B Include the index of the beginning of the match in the result.

E Include the index one character past the end of the match in the result (note this is inconsistent with other uses of parameter index).

M Include the matched portion in the result.

N Include the length of the match in the result.

R Include the unmatched portion in the result (the *R*est).

Rules

Here is a summary of the rules for substitution; this assumes that braces are present around the substitution, i.e. **${**...**}**. Some particular examples are given below. Note that the Zsh Development Group accepts *no responsibility* for any brain damage which may occur during the reading of the following rules.

1. *Nested substitution*

If multiple nested **${**...**}** forms are present, substitution is performed from the inside outwards. At each level, the substitution takes account of whether the current value is a scalar or an array, whether the whole substitution is in double quotes, and what flags are supplied to the current level of substitution, just as if the nested substitution were the outermost. The flags are not propagated up to enclosing substitutions; the nested substitution will return either a scalar or an array as determined by the flags, possibly adjusted for quoting. All the following steps take place where applicable at all levels of substitution.

Note that, unless the '**(P)**' flag is present, the flags and any subscripts apply directly to the value of the nested substitution; for example, the expansion **${${foo}}** behaves exactly the same as **${foo}**. When the '**(P)**' flag is present in a nested substitution, the other substitution rules are applied to the value *before* it is interpreted as a name, so **${${(P)foo}}** may differ from **${(P)foo}**.

At each nested level of substitution, the substituted words undergo all forms of single−word substitution (i.e. not filename generation), including command substitution, arithmetic expansion and filename expansion (i.e. leading ~ and =). Thus, for example, **${${:−=cat}:h}** expands to the directory where the **cat** program resides. (Explanation: the internal substitution has no parameter but a default value **=cat**, which is expanded by filename expansion to a full path; the outer substitution then applies the modifier **:h** and takes the directory part of the path.)

2. *Internal parameter flags*

Any parameter flags set by one of the **typeset** family of commands, in particular the **−L**, **−R**, **−Z**, **−u** and **−l** options for padding and capitalization, are applied directly to the parameter value. Note these flags are options to the command, e.g. '**typeset −Z**'; they are not the same as the flags used within parameter substitutions.

At the outermost level of substitution, the '**(P)**' flag (rule **4.**) ignores these transformations and uses the unmodified value of the parameter as the name to be replaced. This is usually the desired behavior because padding may make the value syntactically illegal as a parameter name, but if capitalization changes are desired, use the **${${(P)foo}}** form (rule **25.**).

3. *Parameter subscripting*

If the value is a raw parameter reference with a subscript, such as **${***var***[3]}**, the effect of subscripting is applied directly to the parameter. Subscripts are evaluated left to right; subsequent subscripts apply to the scalar or array value yielded by the previous subscript. Thus if **var** is an array, **${var[1][2]}** is the second character of the first word, but **${var[2,4][2]}** is the entire third word (the second word of the range of words two through four of the original array). Any number of subscripts may appear. Flags such as '**(k)**' and '**(v)**' which alter the result of subscripting are applied.

4. *Parameter name replacement*

At the outermost level of nesting only, the '**(P)**' flag is applied. This treats the value so far as a parameter name (which may include a subscript expression) and replaces that with the corresponding value. This replacement occurs later if the '**(P)**' flag appears in a nested substitution.

If the value so far names a parameter that has internal flags (rule **2.**), those internal flags are applied to the new value after replacement.

5. *Double−quoted joining*

If the value after this process is an array, and the substitution appears in double quotes, and neither an '**(@)**' flag nor a '**#**' length operator is present at the current level, then words of the value are joined with the first character of the parameter **$IFS**, by default a space, between each word (single word arrays are not modified). If the '**(j)**' flag is present, that is used for joining instead of **$IFS**.

6. *Nested subscripting*

Any remaining subscripts (i.e. of a nested substitution) are evaluated at this point, based on whether the value is an array or a scalar. As with **3.**, multiple subscripts can appear. Note that **${foo[2,4][2]}** is thus equivalent to **${${foo[2,4]}[2]}** and also to **"${${(@)foo[2,4]}[2]}"** (the nested substitution returns an array in both cases), but not to **"${${foo[2,4]}[2]}"** (the nested substitution returns a scalar because of the quotes).

7. *Modifiers*

Any modifiers, as specified by a trailing '**#**', '**%**', '/' (possibly doubled) or by a set of modifiers of the form '**:...**' (see the section 'Modifiers' in the section 'History Expansion'), are applied to the words of the value at this level.

8. *Character evaluation*

Any '**(#)**' flag is applied, evaluating the result so far numerically as a character.

9. *Length*

Any initial '**#**' modifier, i.e. in the form **${#***var***}**, is used to evaluate the length of the expression so far.

10. *Forced joining*

If the '**(j)**' flag is present, or no '**(j)**' flag is present but the string is to be split as given by rule **11.**, and joining did not take place at rule **5.**, any words in the value are joined together using the given string or the first character of **$IFS** if none. Note that the '**(F)**' flag implicitly supplies a string for joining in this manner.

11. *Simple word splitting*
If one of the '**(s)**' or '**(f)**' flags are present, or the '**=**' specifier was present (e.g. **${=***var***}**), the word is split on occurrences of the specified string, or (for **=** with neither of the two flags present) any of the characters in **$IFS**.

If no '**(s)**', '**(f)**' or '**=**' was given, but the word is not quoted and the option **SH_WORD_SPLIT** is set, the word is split on occurrences of any of the characters in **$IFS**. Note this step, too, takes place at all levels of a nested substitution.

12. *Case modification*
Any case modification from one of the flags '**(L)**', '**(U)**' or '**(C)**' is applied.

13. *Escape sequence replacement*
First any replacements from the '**(g)**' flag are performed, then any prompt−style formatting from the '**(%)**' family of flags is applied.

14. *Quote application*
Any quoting or unquoting using '**(q)**' and '**(Q)**' and related flags is applied.

15. *Directory naming*
Any directory name substitution using '**(D)**' flag is applied.

16. *Visibility enhancement*
Any modifications to make characters visible using the '**(V)**' flag are applied.

17. *Lexical word splitting*
If the '**(z)**' flag or one of the forms of the '**(Z)**' flag is present, the word is split as if it were a shell command line, so that quotation marks and other metacharacters are used to decide what constitutes a word. Note this form of splitting is entirely distinct from that described by rule**11.**: it does not use **$IFS**, and does not cause forced joining.

18. *Uniqueness*
If the result is an array and the '**(u)**' flag was present, duplicate elements are removed from the array.

19. *Ordering*
If the result is still an array and one of the '**(o)**' or '**(O)**' flags was present, the array is reordered.

20. RC_EXPAND_PARAM
At this point the decision is made whether any resulting array elements are to be combined element by element with surrounding text, as given by either the **RC_EXPAND_PARAM** option or the '**^**' flag.

21. *Re−evaluation*
Any '**(e)**' flag is applied to the value, forcing it to be re−examined for new parameter substitutions, but also for command and arithmetic substitutions.

22. *Padding*
Any padding of the value by the '**(l.***fill***.)**' or '**(r.***fill***.)**' flags is applied.

23. *Semantic joining*
In contexts where expansion semantics requires a single word to result, all words are rejoined with the first character of **IFS** between. So in '**${(P)${(f)lines}}**' the value of **${lines}** is split at newlines, but then must be joined again before the '**(P)**' flag can be applied.

If a single word is not required, this rule is skipped.

24. *Empty argument removal*
If the substitution does not appear in double quotes, any resulting zero−length argument, whether from a scalar or an element of an array, is elided from the list of arguments inserted into the command line.

Strictly speaking, the removal happens later as the same happens with other forms of substitution; the point to note here is simply that it occurs after any of the above parameter operations.

25. *Nested parameter name replacement*

If the '**(P)**' flag is present and rule **4.** has not applied, the value so far is treated as a parameter name (which may include a subscript expression) and replaced with the corresponding value, with internal flags (rule **2.**) applied to the new value.

Examples

The flag **f** is useful to split a double−quoted substitution line by line. For example, **${(f)"$(<***file***)"}** substitutes the contents of *file* divided so that each line is an element of the resulting array. Compare this with the effect of **$(<***file***)** alone, which divides the file up by words, or the same inside double quotes, which makes the entire content of the file a single string.

The following illustrates the rules for nested parameter expansions. Suppose that**$f oo** contains the array **(bar baz)**:

"${(@)${foo}[1]}"

This produces the result **b**. First, the inner substitution**"${f oo}"**, which has no array (**@**) flag, produces a single word result **"bar baz"**. The outer substitution**"${(@)...[1]}"** detects that this is a scalar, so that (despite the '**(@)**' flag) the subscript picks the first character.

"${${(@)foo}[1]}"

This produces the result '**bar**'. In this case, the inner substitution**"${(@)f oo}"** produces the array '**(bar baz)**'. The outer substitution**"${...[1]}"** detects that this is an array and picks the first w ord. This is similar to the simple case **"${foo[1]}"**.

As an example of the rules for word splitting and joining, suppose **$foo** contains the array '**(ax1 bx1)**'. Then

${(s/x/)foo}

produces the words '**a**', '**1 b**' and '**1**'.

${(j/x/s/x/)foo}

produces '**a**', '**1**', '**b**' and '**1**'.

${(s/x/)foo%%1*}

produces '**a**' and ' **b**' (note the extra space). As substitution occurs before either joining or splitting, the operation first generates the modified array **(ax bx)**, which is joined to give **"ax bx"**, and then split to give '**a**', ' **b**' and ''. The final empty string will then be elided, as it is not in double quotes.

COMMAND SUBSTITUTION

A command enclosed in parentheses preceded by a dollar sign, like '**$(**...**)**', or quoted with grave accents, like '**`**...**`**', is replaced with its standard output, with any trailing newlines deleted. If the substitution is not enclosed in double quotes, the output is broken into words using the **IFS** parameter.

The substitution '**$(cat** *foo***)**' may be replaced by the faster '**$(<***foo***)**'. In this case*foo* under goes single word shell expansions (*parameter expansion*, *command substitution* and *arithmetic expansion*), but not filename generation.

If the option **GLOB_SUBST** is set, the result of any unquoted command substitution, including the special form just mentioned, is eligible for filename generation.

ARITHMETIC EXPANSION

A string of the form '**$[***exp***]**' or '**$((***exp***))**' is substituted with the value of the arithmetic expression *exp*. *exp* is subjected to *parameter expansion*, *command substitution* and *arithmetic expansion* before it is evaluated. See the section 'Arithmetic Evaluation'.

BRACE EXPANSION

A string of the form '*foo***{***xx***,***yy***,***zz***}***bar*' is expanded to the individual words '*fooxxbar*', '*fooyybar*' and '*foozzbar*'. Left−to−right order is preserved. This construct may be nested. Commas may be quoted in order to include them literally in a word.

An expression of the form '**{***n1***..***n2***}**', where *n1* and *n2* are integers, is expanded to every number between *n1* and *n2* inclusive. If either number begins with a zero, all the resulting numbers will be padded with

leading zeroes to that minimum width, but for negative numbers the − character is also included in the width. If the numbers are in decreasing order the resulting sequence will also be in decreasing order.

An expression of the form '**{***n1***..***n2***..***n3***}**', where *n1*, *n2*, and *n3* are integers, is expanded as above, but only every *n3*th number starting from *n1* is output. If *n3* is negative the numbers are output in reverse order, this is slightly different from simply swapping *n1* and *n2* in the case that the step *n3* doesn't evenly divide the range. Zero padding can be specified in any of the three numbers, specifying it in the third can be useful to pad for example '**{−99..100..01}**' which is not possible to specify by putting a 0 on either of the first two numbers (i.e. pad to two characters).

An expression of the form '**{***c1***..***c2***}**', where *c1* and *c2* are single characters (which may be multibyte characters), is expanded to every character in the range from *c1* to *c2* in whatever character sequence is used internally. For characters with code points below 128 this is US ASCII (this is the only case most users will need). If any intervening character is not printable, appropriate quotation is used to render it printable. If the character sequence is reversed, the output is in reverse order, e.g. '**{d..a}**' is substituted as '**d c b a**'.

If a brace expression matches none of the above forms, it is left unchanged, unless the option **BRACE_CCL** (an abbreviation for 'brace character class') is set. In that case, it is expanded to a list of the individual characters between the braces sorted into the order of the characters in the ASCII character set (multibyte characters are not currently handled). The syntax is similar to a **[...]** expression in filename generation: '**−**' is treated specially to denote a range of characters, but '**^**' or '**!**' as the first character is treated normally. For example, '**{abcdef0−9}**' expands to 16 words **0 1 2 3 4 5 6 7 8 9 a b c d e f**.

Note that brace expansion is not part of filename generation (globbing); an expression such as ***/{foo,bar}** is split into two separate words ***/foo** and ***/bar** before filename generation takes place. In particular, note that this is liable to produce a 'no match' error if *either* of the two expressions does not match; this is to be contrasted with ***/(foo|bar)**, which is treated as a single pattern but otherwise has similar effects.

To combine brace expansion with array expansion, see the **${^***spec***}** form described in the section Parameter Expansion above.

FILENAME EXPANSION

Each word is checked to see if it begins with an unquoted '**~**'. If it does, then the word up to a '/', or the end of the word if there is no '/', is checked to see if it can be substituted in one of the ways described here. If so, then the '**~**' and the checked portion are replaced with the appropriate substitute value.

A '**~**' by itself is replaced by the value of **$HOME**. A '**~**' followed by a '**+**' or a '**−**' is replaced by current or previous working directory, respectively.

A '**~**' followed by a number is replaced by the directory at that position in the directory stack. '**~0**' is equivalent to '**~+**', and '**~1**' is the top of the stack. '**~+**' followed by a number is replaced by the directory at that position in the directory stack. '**~+0**' is equivalent to '**~+**', and '**~+1**' is the top of the stack. '**~−**' followed by a number is replaced by the directory that many positions from the bottom of the stack. '**~−0**' is the bottom of the stack. The **PUSHD_MINUS** option exchanges the effects of '**~+**' and '**~−**' where they are followed by a number.

Dynamic named directories

If the function **zsh_directory_name** exists, or the shell variable **zsh_directory_name_functions** exists and contains an array of function names, then the functions are used to implement dynamic directory naming. The functions are tried in order until one returns status zero, so it is important that functions test whether they can handle the case in question and return an appropriate status.

A '**~**' followed by a string *namstr* in unquoted square brackets is treated specially as a dynamic directory name. Note that the first unquoted closing square bracket always terminates *namstr*. The shell function is passed two arguments: the string **n** (for name) and *namstr*. It should either set the array**r eply** to a single element which is the directory corresponding to the name and return status zero (executing an assignment as the last statement is usually sufficient), or it should return status non−zero. In the former case the element of reply is used as the directory; in the latter case the substitution is deemed to have failed. If all functions fail and the option **NOMATCH** is set, an error results.

The functions defined as above are also used to see if a directory can be turned into a name, for example

when printing the directory stack or when expanding **%~** in prompts. In this case each function is passed two arguments: the string **d** (for directory) and the candidate for dynamic naming. The function should either return non−zero status, if the directory cannot be named by the function, or it should set the array reply to consist of two elements: the first is the dynamic name for the directory (as would appear within '**~[**...**]**'), and the second is the prefix length of the directory to be replaced. For example, if the trial directory is **/home/myname/src/zsh** and the dynamic name for **/home/myname/src** (which has 16 characters) is **s**, then the function sets

reply=(s 16)

The directory name so returned is compared with possible static names for parts of the directory path, as described below; it is used if the prefix length matched (16 in the example) is longer than that matched by any static name.

It is not a requirement that a function implements both **n** and **d** calls; for example, it might be appropriate for certain dynamic forms of expansion not to be contracted to names. In that case any call with the first argument **d** should cause a non−zero status to be returned.

The completion system calls '**zsh_directory_name c**' followed by equivalent calls to elements of the array **zsh_directory_name_functions**, if it exists, in order to complete dynamic names for directories. The code for this should be as for any other completion function as described in *zshcompsys*(1).

As a working example, here is a function that expands any dynamic names beginning with the string **p:** to directories below **/home/pws/perforce**. In this simple case a static name for the directory would be just as effective.

```
zsh_directory_name() {
  emulate −L zsh
  setopt extendedglob
  local −a match mbegin mend
  if [[ $1 = d ]]; then
    # turn the directory into a name
    if [[ $2 = (#b)(/home/pws/perforce/)([^/]##)* ]]; then
      typeset −ga reply
      reply=(p:$match[2] $(( ${#match[1]} + ${#match[2]} )) )
    else
      return 1
    fi
  elif [[ $1 = n ]]; then
    # turn the name into a directory
    [[ $2 != (#b)p:(?*) ]] && return 1
    typeset −ga reply
    reply=(/home/pws/perforce/$match[1])
  elif [[ $1 = c ]]; then
    # complete names
    local expl
    local −a dirs
    dirs=(/home/pws/perforce/*(/:t))
    dirs=(p:${^dirs})
    _wanted dynamic−dirs expl 'dynamic directory' compadd −S\] −a dirs
    return
  else
    return 1
  fi
  return 0
}
```

Static named directories

A '**~**' followed by anything not already covered consisting of any number of alphanumeric characters or underscore ('**_**'), hyphen ('**-**'), or dot ('**.**') is looked up as a named directory, and replaced by the value of that named directory if found. Named directories are typically home directories for users on the system. They may also be defined if the text after the '**~**' is the name of a string shell parameter whose value begins with a '/'. Note that trailing slashes will be removed from the path to the directory (though the original parameter is not modified).

It is also possible to define directory names using the **-d** option to the **hash** builtin.

When the shell prints a path (e.g. when expanding **%~** in prompts or when printing the directory stack), the path is checked to see if it has a named directory as its prefix. If so, then the prefix portion is replaced with a '**~**' followed by the name of the directory. The shorter of the two ways of referring to the directory is used, i.e. either the directory name or the full path; the name is used if they are the same length. The parameters **$PWD** and **$OLDPWD** are never abbreviated in this fashion.

'=' expansion

If a word begins with an unquoted '**=**' and the **EQUALS** option is set, the remainder of the word is taken as the name of a command. If a command exists by that name, the word is replaced by the full pathname of the command.

Notes

Filename expansion is performed on the right hand side of a parameter assignment, including those appearing after commands of the **typeset** family. In this case, the right hand side will be treated as a colon-separated list in the manner of the **PATH** parameter, so that a '**~**' or an '**=**' following a '**:**' is eligible for expansion. All such behaviour can be disabled by quoting the '**~**', the '**=**', or the whole expression (but not simply the colon); the **EQUALS** option is also respected.

If the option **MAGIC_EQUAL_SUBST** is set, any unquoted shell argument in the form '*identifier*=*expression*' becomes eligible for file expansion as described in the previous paragraph. Quoting the first '**=**' also inhibits this.

FILENAME GENERATION

If a word contains an unquoted instance of one of the characters '*****', '**(**', '**|**', '**<**', '**[**', or '**?**', it is regarded as a pattern for filename generation, unless the **GLOB** option is unset. If the **EXTENDED_GLOB** option is set, the '**^**' and '**#**' characters also denote a pattern; otherwise they are not treated specially by the shell.

The word is replaced with a list of sorted filenames that match the pattern. If no matching pattern is found, the shell gives an error message, unless the **NULL_GLOB** option is set, in which case the word is deleted; or unless the **NOMATCH** option is unset, in which case the word is left unchanged.

In filename generation, the character '/' must be matched explicitly; also, a '**.**' must be matched explicitly at the beginning of a pattern or after a '/', unless the **GLOB_DOTS** option is set. No filename generation pattern matches the files '**.**' or '**..**'. In other instances of pattern matching, the '/' and '**.**' are not treated specially.

Glob Operators

***** Matches any string, including the null string.

? Matches any character.

[...] Matches any of the enclosed characters. Ranges of characters can be specified by separating two characters by a '**-**'. A '**-**' or '**]**' may be matched by including it as the first character in the list. There are also several named classes of characters, in the form '**[:***name***:]**' with the following meanings. The first set use the macros provided by the operating system to test for the given character combinations, including any modifications due to local language settings, see *ctype*(3):

[:alnum:]

The character is alphanumeric

[:alpha:]
The character is alphabetic

[:ascii:]
The character is 7−bit, i.e. is a single−byte character without the top bit set.

[:blank:]
The character is a blank character

[:cntrl:]
The character is a control character

[:digit:]
The character is a decimal digit

[:graph:]
The character is a printable character other than whitespace

[:lower:]
The character is a lowercase letter

[:print:]
The character is printable

[:punct:]
The character is printable but neither alphanumeric nor whitespace

[:space:]
The character is whitespace

[:upper:]
The character is an uppercase letter

[:xdigit:]
The character is a hexadecimal digit

Another set of named classes is handled internally by the shell and is not sensitive to the locale:

[:IDENT:]
The character is allowed to form part of a shell identifier, such as a parameter name

[:IFS:] The character is used as an input field separator, i.e. is contained in the **IFS** parameter

[:IFSSPACE:]
The character is an IFS white space character; see the documentation for **IFS** in the *zshparam*(1) manual page.

[:INCOMPLETE:]
Matches a byte that starts an incomplete multibyte character. Note that there may be a sequence of more than one bytes that taken together form the prefix of a multibyte character. To test for a potentially incomplete byte sequence, use the pattern '**[[:INCOMPLETE:]]***'. This will never match a sequence starting with a valid multibyte character.

[:INVALID:]
Matches a byte that does not start a valid multibyte character. Note this may be a continuation byte of an incomplete multibyte character as any part of a multibyte string consisting of invalid and incomplete multibyte characters is treated as single bytes.

[:WORD:]
The character is treated as part of a word; this test is sensitive to the value of the **WORDCHARS** parameter

Note that the square brackets are additional to those enclosing the whole set of characters, so to test for a single alphanumeric character you need '**[[:alnum:]]**'. Named character sets can be used alongside other types, e.g. '**[[:alpha:]0−9]**'.

[^...]
[!...] Like **[...]**, except that it matches any character which is not in the given set.

<[*x*]−[*y*]>
Matches any number in the range *x* to *y*, inclusive. Either of the numbers may be omitted to make the range open−ended; hence '**<−>**' matches any number. To match individual digits, the **[...]** form is more efficient.

Be careful when using other wildcards adjacent to patterns of this form; for example, **<0−9>*** will actually match any number whatsoever at the start of the string, since the '**<0−9>**' will match the first digit, and the '*****' will match any others. This is a trap for the unwary, but is in fact an inevitable consequence of the rule that the longest possible match always succeeds. Expressions such as '**<0−9>[^[:digit:]]***' can be used instead.

(...**)** Matches the enclosed pattern. This is used for grouping. If the **KSH_GLOB** option is set, then a '**@**', '*****', '**+**', '**?**' or '**!**' immediately preceding the '**(**' is treated specially, as detailed below. The option **SH_GLOB** prevents bare parentheses from being used in this way, though the **KSH_GLOB** option is still available.

Note that grouping cannot extend over multiple directories: it is an error to have a '/' within a group (this only applies for patterns used in filename generation). There is one exception: a group of the form **(***pat***/)#** appearing as a complete path segment can match a sequence of directories. For example, **foo/(a*/)#bar** matches **foo/bar**, **foo/any/bar**, **foo/any/anyother/bar**, and so on.

x|*y* Matches either *x* or *y*. This operator has lower precedence than any other. The '|' character must be within parentheses, to avoid interpretation as a pipeline. The alternatives are tried in order from left to right.

^*x* (Requires **EXTENDED_GLOB** to be set.) Matches anything except the pattern *x*. This has a higher precedence than '/', so '**^foo/bar**' will search directories in '**.**' except '**./foo**' for a file named '**bar**'.

x~*y* (Requires **EXTENDED_GLOB** to be set.) Match anything that matches the pattern *x* but does not match *y*. This has lower precedence than any operator except '|', so '***/*~foo/bar**' will search for all files in all directories in '**.**' and then exclude '**foo/bar**' if there was such a match. Multiple patterns can be excluded by '*foo*~*bar*~*baz*'. In the exclusion pattern (*y*), '/' and '**.**' are not treated specially the way they usually are in globbing.

x# (Requires **EXTENDED_GLOB** to be set.) Matches zero or more occurrences of the pattern *x*. This operator has high precedence; '**12#**' is equivalent to '**1(2#)**', rather than '**(12)#**'. It is an error for an unquoted '**#**' to follow something which cannot be repeated; this includes an empty string, a pattern already followed by '**##**', or parentheses when part of a **KSH_GLOB** pattern (for example, '**!(***foo***)#**' is invalid and must be replaced by '***(!(***foo***))**').

x## (Requires **EXTENDED_GLOB** to be set.) Matches one or more occurrences of the pattern *x*. This operator has high precedence; '**12##**' is equivalent to '**1(2##)**', rather than '**(12)##**'. No more than two active '**#**' characters may appear together. (Note the potential clash with glob qualifiers in the form '**1(2##)**' which should therefore be avoided.)

ksh−like Glob Operators

If the **KSH_GLOB** option is set, the effects of parentheses can be modified by a preceding '**@**', '*****', '**+**', '**?**' or '**!**'. This character need not be unquoted to have special effects, but the '**(**' must be.

@(...**)** Match the pattern in the parentheses. (Like '**(**...**)**'.)

***(**...**)** Match any number of occurrences. (Like '**(**...**)#**', except that recursive directory searching is not supported.)

+(...**)** Match at least one occurrence. (Like '**(**...**)##**', except that recursive directory searching is not supported.)

?(...**)** Match zero or one occurrence. (Like '**(**|...**)**'.)

!(...) Match anything but the expression in parentheses. (Like '**(^(...))**'.)

Precedence

The precedence of the operators given above is (highest) '^', '/', '~', '|' (lowest); the remaining operators are simply treated from left to right as part of a string, with '**#**' and '**##**' applying to the shortest possible preceding unit (i.e. a character, '**?**', '**[...]**', '<...>', or a parenthesised expression). As mentioned above, a '/' used as a directory separator may not appear inside parentheses, while a '|' must do so; in patterns used in other contexts than filename generation (for example, in **case** statements and tests within '**[[...]]**'), a '/' is not special; and '/' is also not special after a '**~**' appearing outside parentheses in a filename pattern.

Globbing Flags

There are various flags which affect any text to their right up to the end of the enclosing group or to the end of the pattern; they require the **EXTENDED_GLOB** option. All take the form **(#***X***)** where *X* may have one of the following forms:

i Case insensitive: upper or lower case characters in the pattern match upper or lower case characters.

l Lower case characters in the pattern match upper or lower case characters; upper case characters in the pattern still only match upper case characters.

I Case sensitive: locally negates the effect of **i** or **l** from that point on.

b Activate backreferences for parenthesised groups in the pattern; this does not work in filename generation. When a pattern with a set of active parentheses is matched, the strings matched by the groups are stored in the array **$match**, the indices of the beginning of the matched parentheses in the array **$mbegin**, and the indices of the end in the array **$mend**, with the first element of each array corresponding to the first parenthesised group, and so on. These arrays are not otherwise special to the shell. The indices use the same convention as does parameter substitution, so that elements of **$mend** and **$mbegin** may be used in subscripts; the **KSH_ARRAYS** option is respected. Sets of globbing flags are not considered parenthesised groups; only the first nine active parentheses can be referenced.

For example,

```
foo="a_string_with_a_message"
if [[ $foo = (a|an)_(#b)(*) ]]; then
  print ${foo[$mbegin[1],$mend[1]]}
fi
```

prints '**string_with_a_message**'. Note that the first set of parentheses is before the**(#b)** and does not create a backreference.

Backreferences work with all forms of pattern matching other than filename generation, but note that when performing matches on an entire array, such as **${***array***#***pattern***}**, or a global substitution, such as **${***param***//***pat***/***repl***}**, only the data for the last match remains available. In the case of global replacements this may still be useful. See the example for the **m** flag below.

The numbering of backreferences strictly follows the order of the opening parentheses from left to right in the pattern string, although sets of parentheses may be nested. There are special rules for parentheses followed by '**#**' or '**##**'. Only the last match of the parenthesis is remembered: for example, in '**[[abab = (#b)([ab])#]]**', only the final '**b**' is stored in **match[1]**. Thus extra parentheses may be necessary to match the complete segment: for example, use '**X((ab|cd)#)Y**' to match a whole string of either '**ab**' or '**cd**' between '**X**' and '**Y**', using the value of **$match[1]** rather than **$match[2]**.

If the match fails none of the parameters is altered, so in some cases it may be necessary to initialise them beforehand. If some of the backreferences fail to match −− which happens if they are in an alternate branch which fails to match, or if they are followed by # and matched zero times −− then the matched string is set to the empty string, and the start and end indices are set to −1.

Pattern matching with backreferences is slightly slower than without.

B Deactivate backreferences, negating the effect of the **b** flag from that point on.

c*N*,*M* The flag **(#c***N*,*M***)** can be used anywhere that the # or ## operators can be used except in the expressions '**(*/)#**' and '**(*/)##**' in filename generation, where '/' has special meaning; it cannot be combined with other globbing flags and a bad pattern error occurs if it is misplaced. It is equivalent to the form **{***N*,*M***}** in regular expressions. The previous character or group is required to match between *N* and *M* times, inclusive. The form**(#c***N***)** requires e xactly **N** matches; **(#c,***M***)** is equivalent to specifying *N* as 0; **(#c***N***,)** specifies that there is no maximum limit on the number of matches.

m Set references to the match data for the entire string matched; this is similar to backreferencing and does not work in filename generation. The flag must be in effect at the end of the pattern, i.e. not local to a group. The parameters **$MATCH**, **$MBEGIN** and **$MEND** will be set to the string matched and to the indices of the beginning and end of the string, respectively. This is most useful in parameter substitutions, as otherwise the string matched is obvious.

For example,

```
arr=(veldt jynx grimps waqf zho buck)
print ${arr//(#m)[aeiou]/${(U)MATCH}}
```

forces all the matches (i.e. all vowels) into uppercase, printing '**vEldt jynx grImps wAqf zhO bUck**'.

Unlike backreferences, there is no speed penalty for using match references, other than the extra substitutions required for the replacement strings in cases such as the example shown.

M Deactivate the **m** flag, hence no references to match data will be created.

a*num* Approximate matching: *num* errors are allowed in the string matched by the pattern. The rules for this are described in the next subsection.

s, **e** Unlike the other flags, these have only a local effect, and each must appear on its own: '**(#s)**' and '**(#e)**' are the only valid forms. The '**(#s)**' flag succeeds only at the start of the test string, and the '**(#e)**' flag succeeds only at the end of the test string; they correspond to '^' and '**$**' in standard regular expressions. They are useful for matching path segments in patterns other than those in filename generation (where path segments are in any case treated separately). For example, '***((#s)|/)test((#e)|/)***' matches a path segment '**test**' in any of the following strings: **test**, **test/at/start**, **at/end/test**, **in/test/middle**.

Another use is in parameter substitution; for example '**${array/(#s)A*Z(#e)}**' will remove only elements of an array which match the complete pattern '**A*Z**'. There are other ways of performing many operations of this type, however the combination of the substitution operations '/' and '//' with the '**(#s)**' and '**(#e)**' flags provides a single simple and memorable method.

Note that assertions of the form '**(^(#s))**' also work, i.e. match anywhere except at the start of the string, although this actually means 'anything except a zero-length portion at the start of the string'; you need to use '**("")~(#s))**' to match a zero-length portion of the string not at the start.

q A '**q**' and everything up to the closing parenthesis of the globbing flags are ignored by the pattern matching code. This is intended to support the use of glob qualifiers, see below. The result is that the pattern '**(#b)(*).c(#q.)**' can be used both for globbing and for matching against a string. In the former case, the '**(#q.)**' will be treated as a glob qualifier and the '**(#b)**' will not be useful, while in the latter case the '**(#b)**' is useful for backreferences and the '**(#q.)**' will be ignored. Note that colon modifiers in the glob qualifiers are also not applied in ordinary pattern matching.

u Respect the current locale in determining the presence of multibyte characters in a pattern, provided the shell was compiled with **MULTIBYTE_SUPPORT**. This overrides the **MULTIBYTE** option; the default behaviour is taken from the option. Compare **U**. (Mnemonic: typically multibyte characters are from Unicode in the UTF-8 encoding, although any extension of ASCII supported by the system library may be used.)

U All characters are considered to be a single byte long. The opposite of **u**. This overrides the **MULTIBYTE** option.

For example, the test string **fooxx** can be matched by the pattern **(#i)FOOXX**, but not by **(#l)FOOXX**, **(#i)FOO(#I)XX** or **((#i)FOOX)X**. The string**(#ia2)r eadme** specifies case−insensitive matching of **readme** with up to two errors.

When using the ksh syntax for grouping both **KSH_GLOB** and **EXTENDED_GLOB** must be set and the left parenthesis should be preceded by **@**. Note also that the flags do not affect letters inside **[...]** groups, in other words **(#i)[a−z]** still matches only lowercase letters. Finally, note that when examining whole paths case−insensitively every directory must be searched for all files which match, so that a pattern of the form **(#i)/foo/bar/...** is potentially slow.

Approximate Matching

When matching approximately, the shell keeps a count of the errors found, which cannot exceed the number specified in the **(#a***num***)** flags. Four types of error are recognised:

1. Different characters, as in **fooxbar** and **fooybar**.
2. Transposition of characters, as in **banana** and **abnana**.
3. A character missing in the target string, as with the pattern **road** and target string **rod**.
4. An extra character appearing in the target string, as with **stove** and **strove**.

Thus, the pattern **(#a3)abcd** matches **dcba**, with the errors occurring by using the first rule twice and the second once, grouping the string as **[d][cb][a]** and **[a][bc][d]**.

Non−literal parts of the pattern must match exactly, including characters in character ranges: hence **(#a1)???** matches strings of length four, by applying rule 4 to an empty part of the pattern, but not strings of length two, since all the **?** must match. Other characters which must match exactly are initial dots in filenames (unless the **GLOB_DOTS** option is set), and all slashes in filenames, so that **a/bc** is two errors from **ab/c** (the slash cannot be transposed with another character). Similarly, errors are counted separately for non−contiguous strings in the pattern, so that **(ab|cd)ef** is two errors from **aebf**.

When using exclusion via the **~** operator, approximate matching is treated entirely separately for the excluded part and must be activated separately. Thus, **(#a1)README~READ_ME** matches **READ.ME** but not **READ_ME**, as the trailing **READ_ME** is matched without approximation. However, **(#a1)README~(#a1)READ_ME** does not match any pattern of the form **READ***?***ME** as all such forms are now excluded.

Apart from exclusions, there is only one overall error count; however, the maximum errors allowed may be altered locally, and this can be delimited by grouping. For example, **(#a1)cat((#a0)dog)fox** allows one error in total, which may not occur in the **dog** section, and the pattern **(#a1)cat(#a0)dog(#a1)fox** is equivalent. Note that the point at which an error is first found is the crucial one for establishing whether to use approximation; for example, **(#a1)abc(#a0)xyz** will not match **abcdxyz**, because the error occurs at the '**x**', where approximation is turned off.

Entire path segments may be matched approximately, so that '**(#a1)/foo/d/is/available/at/the/bar**' allows one error in any path segment. This is much less efficient than without the **(#a1)**, however, since every directory in the path must be scanned for a possible approximate match. It is best to place the **(#a1)** after any path segments which are known to be correct.

Recursive Globbing

A pathname component of the form '**(***foo***/)#**' matches a path consisting of zero or more directories matching the pattern *foo*.

As a shorthand, '****/**' is equivalent to '**(*/)#**'; note that this therefore matches files in the current directory as well as subdirectories. Thus:

ls −ld −− (*/)#bar

or

ls −ld −− **/bar

does a recursive directory search for files named '**bar**' (potentially including the file '**bar**' in the current directory). This form does not follow symbolic links; the alternative form '*******/' does, but is otherwise identical. Neither of these can be combined with other forms of globbing within the same path segment; in that case, the '*****' operators revert to their usual effect.

Even shorter forms are available when the option **GLOB_STAR_SHORT** is set. In that case if no / immediately follows a ****** or ******* they are treated as if both a / plus a further ***** are present. Hence:

setopt GLOBSTARSHORT
ls −ld −− **.c

is equivalent to

ls −ld −− **/*.c

Glob Qualifiers

Patterns used for filename generation may end in a list of qualifiers enclosed in parentheses. The qualifiers specify which filenames that otherwise match the given pattern will be inserted in the argument list.

If the option **BARE_GLOB_QUAL** is set, then a trailing set of parentheses containing no '|' or '**(**' characters (or '**~**' if it is special) is taken as a set of glob qualifiers. A glob subexpression that would normally be taken as glob qualifiers, for example '**(^x)**', can be forced to be treated as part of the glob pattern by doubling the parentheses, in this case producing '**((^x))**'.

If the option **EXTENDED_GLOB** is set, a different syntax for glob qualifiers is available, namely '**(#q***x***)**' where *x* is any of the same glob qualifiers used in the other format. The qualifiers must still appear at the end of the pattern. However, with this syntax multiple glob qualifiers may be chained together. They are treated as a logical AND of the individual sets of flags. Also, as the syntax is unambiguous, the expression will be treated as glob qualifiers just as long any parentheses contained within it are balanced; appearance of '|', '**(**' or '**~**' does not negate the effect. Note that qualifiers will be recognised in this form even if a bare glob qualifier exists at the end of the pattern, for example '***(#q*)(.)**' will recognise executable regular files if both options are set; however, mixed syntax should probably be avoided for the sake of clarity. Note that within conditions using the '**[[**' form the presence of a parenthesised expression **(#q**...**)** at the end of a string indicates that globbing should be performed; the expression may include glob qualifiers, but it is also valid if it is simply **(#q)**. This does not apply to the right hand side of pattern match operators as the syntax already has special significance.

A qualifier may be any one of the following:

/ directories

F 'full' (i.e. non−empty) directories. Note that the opposite sense **(^F)** expands to empty directories and all non−directories. Use **(/^F)** for empty directories.

. plain files

@ symbolic links

= sockets

p named pipes (FIFOs)

***** executable plain files (0100 or 0010 or 0001)

% device files (character or block special)

%b block special files

%c character special files

r owner−readable files (0400)

w owner−writable files (0200)

x owner−executable files (0100)

A group−readable files (0040)

I group−writable files (0020)

E group−executable files (0010)

R world−readable files (0004)

W world−writable files (0002)

X world−executable files (0001)

s setuid files (04000)

S setgid files (02000)

t files with the sticky bit (01000)

f*spec* files with access rights matching *spec*. This *spec* may be a octal number optionally preceded by a '=', a '+', or a '−'. If none of these characters is given, the behavior is the same as for '='. The octal number describes the mode bits to be expected, if combined with a '=', the value given must match the file−modes exactly, with a '+', at least the bits in the given number must be set in the file−modes, and with a '−', the bits in the number must not be set. Giving a '**?**' instead of a octal digit anywhere in the number ensures that the corresponding bits in the file−modes are not checked, this is only useful in combination with '='.

If the qualifier '**f**' is followed by any other character anything up to the next matching character ('**[**', '**{**', and '<' match '**]**', '**}**', and '>' respectively, any other character matches itself) is taken as a list of comma−separated *sub−spec*s. Each *sub−spec* may be either an octal number as described above or a list of any of the characters '**u**', '**g**', '**o**', and '**a**', followed by a '=', a '+', or a '−', followed by a list of any of the characters '**r**', '**w**', '**x**', '**s**', and '**t**', or an octal digit. The first list of characters specify which access rights are to be checked. If a '**u**' is given, those for the owner of the file are used, if a '**g**' is given, those of the group are checked, a '**o**' means to test those of other users, and the '**a**' says to test all three groups. The '=', '+', and '−' again says how the modes are to be checked and have the same meaning as described for the first form above. The second list of characters finally says which access rights are to be expected: '**r**' for read access, '**w**' for write access, '**x**' for the right to execute the file (or to search a directory), '**s**' for the setuid and setgid bits, and '**t**' for the sticky bit.

Thus, '***(f70?)**' gives the files for which the owner has read, write, and execute permission, and for which other group members have no rights, independent of the permissions for other users. The pattern '***(f−100)**' gives all files for which the owner does not have execute permission, and '***(f:gu+w,o−rx:)**' gives the files for which the owner and the other members of the group have at least write permission, and for which other users don't have read or execute permission.

e*string*
+*cmd* The *string* will be executed as shell code. The filename will be included in the list if and only if the code returns a zero status (usually the status of the last command).

In the first form, the first character after the '**e**' will be used as a separator and anything up to the next matching separator will be taken as thes*tring*; '**[**', '**{**', and '<' match '**]**', '**}**', and '>', respectively, while any other character matches itself. Note that expansions must be quoted in the *string* to prevent them from being expanded before globbing is done. *string* is then e xecuted as shell code. The string**globqual** is appended to the array **zsh_e val_context** the duration of execution.

During the execution of *string* the filename currently being tested is available in the parameter **REPLY**; the parameter may be altered to a string to be inserted into the list instead of the original filename. In addition, the parameter**r eply** may be set to an array or a string, which overrides the value of **REPLY**. If set to an array, the latter is inserted into the command line word by word.

For example, suppose a directory contains a single file '**lonely**'. Then the expression '***(e:'reply=(${REPLY}{1,2})':)**' will cause the words '**lonely1**' and '**lonely2**' to be inserted into the command line. Note the quoting of *string*.

The form **+***cmd* has the same effect, but no delimiters appear around *cmd*. Instead,*cmd* is tak en as

the longest sequence of characters following the + that are alphanumeric or underscore. Typically *cmd* will be the name of a shell function that contains the appropriate test. For example,

```
nt() { [[ $REPLY −nt $NTREF ]] }
NTREF=reffile
ls −ld −− *(+nt)
```

lists all files in the directory that have been modified more recently than **reffile**.

d*dev* files on the device *dev*

l[−|+]*ct* files having a link count less than *ct* (−), greater than *ct* (+), or equal to *ct*

U files owned by the effective user ID

G files owned by the effective group ID

u*id* files owned by user ID *id* if that is a number. Otherwise, *id* specifies a user name: the character after the '**u**' will be taken as a separator and the string between it and the next matching separator will be taken as a user name. The starting separators '**[**', '**{**', and '<' match the final separators '**]**', '**}**', and '>', respectively; any other character matches itself. The selected files are those owned by this user. For example, '**u:foo:**' or '**u[foo]**' selects files owned by user '**foo**'.

g*id* like **u***id* but with group IDs or names

a[**Mwhms**][−|+]*n*
files accessed exactly *n* days ago. Files accessed within the last *n* days are selected using a negative value for *n* (−*n*). Files accessed more than*n* days ago are selected by a positi ve *n* value (+*n*). Optional unit specifiers '**M**', '**w**', '**h**', '**m**' or '**s**' (e.g. '**ah5**') cause the check to be performed with months (of 30 days), weeks, hours, minutes or seconds instead of days, respectively. An explicit '**d**' for days is also allowed.

Any fractional part of the difference between the access time and the current part in the appropriate units is ignored in the comparison. For instance, '**echo *(ah−5)**' would echo files accessed within the last five hours, while '**echo *(ah+5)**' would echo files accessed at least six hours ago, as times strictly between five and six hours are treated as five hours.

m[**Mwhms**][−|+]*n*
like the file access qualifier, except that it uses the file modification time.

c[**Mwhms**][−|+]*n*
like the file access qualifier, except that it uses the file inode change time.

L[+|−]*n*
files less than *n* bytes (−), more than *n* bytes (+), or exactly *n* bytes in length.

If this flag is directly followed by a *size specifier* '**k**' ('**K**'), '**m**' ('**M**'), or '**p**' ('**P**') (e.g. '**Lk−50**') the check is performed with kilobytes, megabytes, or blocks (of 512 bytes) instead. (On some systems additional specifiers are available for gigabytes, '**g**' or '**G**', and terabytes, '**t**' or '**T**'.) If a size specifier is used a file is regarded as "exactly" the size if the file size rounded up to the next unit is equal to the test size. Hence '***(Lm1)**' matches files from 1 byte up to 1 Megabyte inclusive. Note also that the set of files "less than" the test size only includes files that would not match the equality test; hence '***(Lm−1)**' only matches files of zero size.

^ negates all qualifiers following it

− toggles between making the qualifiers work on symbolic links (the default) and the files they point to

M sets the **MARK_DIRS** option for the current pattern

T appends a trailing qualifier mark to the filenames, analogous to the **LIST_TYPES** option, for the current pattern (overrides **M**)

N sets the **NULL_GLOB** option for the current pattern

D sets the **GLOB_DOTS** option for the current pattern

n sets the **NUMERIC_GLOB_SORT** option for the current pattern

Y*n* enables short−circuit mode: the pattern will expand to at most *n* filenames. If more than *n* matches exist, only the first *n* matches in directory traversal order will be considered.

Implies **oN** when no **o***c* qualifier is used.

o*c* specifies how the names of the files should be sorted. If *c* is **n** they are sorted by name; if it is **L** they are sorted depending on the size (length) of the files; if **l** they are sorted by the number of links; if **a**, **m**, or **c** they are sorted by the time of the last access, modification, or inode change respectively; if **d**, files in subdirectories appear before those in the current directory at each level of the search −− this is best combined with other criteria, for example '**odon**' to sort on names for files within the same directory; if **N**, no sorting is performed. Note that **a**, **m**, and **c** compare the age against the current time, hence the first name in the list is the youngest file. Also note that the modifiers ^ and − are used, so '***(^−oL)**' gives a list of all files sorted by file size in descending order, following any symbolic links. Unless **oN** is used, multiple order specifiers may occur to resolve ties.

The default sorting is **n** (by name) unless the **Y** glob qualifier is used, in which case it is **N** (unsorted).

oe and **o+** are special cases; they are each followed by shell code, delimited as for the **e** glob qualifier and the **+** glob qualifier respectively (see above). The code is executed for each matched file with the parameter **REPLY** set to the name of the file on entry and **globsort** appended to **zsh_eval_context**. The code should modify the parameter**REPL Y** in some fashion. On return, the value of the parameter is used instead of the file name as the string on which to sort. Unlike other sort operators, **oe** and **o+** may be repeated, but note that the maximum number of sort operators of any kind that may appear in any glob expression is 12.

O*c* like '**o**', but sorts in descending order; i.e. '***(^oc)**' is the same as '***(Oc)**' and '***(^Oc)**' is the same as '***(oc)**'; '**Od**' puts files in the current directory before those in subdirectories at each level of the search.

[*beg*[**,***end*]**]**
specifies which of the matched filenames should be included in the returned list. The syntax is the same as for array subscripts. *beg* and the optional *end* may be mathematical expressions. As in parameter subscripting they may be negative to make them count from the last match backward. E.g.: '***(−OL[1,3])**' gives a list of the names of the three largest files.

P*string* The *string* will be prepended to each glob match as a separate word. *string* is delimited in the same way as arguments to the **e** glob qualifier described above. The qualifier can be repeated; the words are prepended separately so that the resulting command line contains the words in the same order they were given in the list of glob qualifiers.

A typical use for this is to prepend an option before all occurrences of a file name; for example, the pattern '***(P:−f:)**' produces the command line arguments '**−f** *file1* **−f** *file2* ...'

If the modifier ^ is active, then *string* will be appended instead of prepended. Prepending and appending is done independently so both can be used on the same glob expression; for example by writing '***(P:foo:^P:bar:^P:baz:)**' which produces the command line arguments '**foo baz** *file1* **bar** ...'

More than one of these lists can be combined, separated by commas. The whole list matches if at least one of the sublists matches (they are 'or'ed, the qualifiers in the sublists are 'and'ed). Some qualifiers, however, affect all matches generated, independent of the sublist in which they are given. These are the qualifiers '**M**', '**T**', '**N**', '**D**', '**n**', '**o**', '**O**' and the subscripts given in brackets ('**[...]**').

If a '**:**' appears in a qualifier list, the remainder of the expression in parenthesis is interpreted as a modifier (see the section 'Modifiers' in the section 'History Expansion'). Each modifier must be introduced by a separate '**:**'. Note also that the result after modification does not have to be an existing file. The name of

any existing file can be followed by a modifier of the form '**(:...)**' even if no actual filename generation is performed, although note that the presence of the parentheses causes the entire expression to be subjected to any global pattern matching options such as **NULL_GLOB**. Thus:

ls −ld −− *(−/)

lists all directories and symbolic links that point to directories, and

ls −ld −− *(−@)

lists all broken symbolic links, and

ls −ld −− *(%W)

lists all world−writable device files in the current directory, and

ls −ld −− *(W,X)

lists all files in the current directory that are world−writable or world−executable, and

print −rC1 /tmp/foo*(u0^@:t)

outputs the basename of all root−owned files beginning with the string '**foo**' in **/tmp**, ignoring symlinks, and

ls −ld −− *.*~(lex|parse).[ch](^D^l1)

lists all files having a link count of one whose names contain a dot (but not those starting with a dot, since **GLOB_DOTS** is explicitly switched off) except for **lex.c**, **lex.h**, **parse.c** and **parse.h**.

print −rC1 b*.pro(#q:s/pro/shmo/)(#q.:s/builtin/shmiltin/)

demonstrates how colon modifiers and other qualifiers may be chained together. The ordinary qualifier '**.**' is applied first, then the colon modifiers in order from left to right. So if **EXTENDED_GLOB** is set and the base pattern matches the regular file **builtin.pro**, the shell will print '**shmiltin.shmo**'.