NAME

csh – shell command interpreter with a C-like syntax

SYNOPSIS

csh [**–bcefinstvVxX**] [*argument*...]

AVAILABILITY

SUNWcsu

DESCRIPTION

csh, the C shell, is a command interpreter with a syntax reminiscent of the C language. It provides a number of convenient features for interactive use that are not available with the Bourne shell, including filename completion, command aliasing, history substitution, job control, and a number of built-in commands. As with the Bourne shell, the C shell provides variable, command and filename substitution.

Initialization and Termination

When first started, the C shell normally performs commands from the **.cshrc** file in your home directory, provided that it is readable and you either own it or your real group ID matches its group ID. If the shell is invoked with a name that starts with '–', as when started by **login**(1), the shell runs as a **login** shell.

If the shell is a login shell, this is the sequence of invocations: First, commands in **/etc/.login** are executed. Next, commands from the **.cshrc** file your **home** directory are executed. Then the shell executes commands from the **.login** file in your home directory; the same permission checks as those for **.cshrc** are applied to this file. Typically, the **.login** file contains commands to specify the terminal type and environment. (For an explanation of file interpreters, see below "Command Execution" and **exec**(2).)

As a login shell terminates, it performs commands from the **.logout** file in your home directory; the same permission checks as those for **.cshrc** are applied to this file.

Interactive Operation

After startup processing is complete, an interactive C shell begins reading commands from the terminal, prompting with *hostname*% (or *hostname*# for the privileged user). The shell then repeatedly performs the following actions: a line of command input is read and broken into *words*. This sequence of words is placed on the history list and then parsed, as described under USAGE, below. Finally, the shell executes each command in the current line.

Noninteractive Operation

When running noninteractively, the shell does not prompt for input from the terminal. A noninteractive C shell can execute a command supplied as an *argument* on its command line, or interpret commands from a file, also known as a script.

OPTIONS

–b Force a "break" from option processing. Subsequent command line arguments are not interpreted as C shell options. This allows the passing of options to a script without confusion. The shell does not run set-user-ID or set-group-ID scripts unless this option is present.

–c Execute the first *argument* (which must be present). Remaining arguments are placed in **argv**, the argument-list variable, and passed directly to **csh**.

–e Exit if a command terminates abnormally or yields a nonzero exit status.

–f Fast start. Read neither the **.cshrc** file, nor the **.login** file (if a login shell) upon startup.

–i Forced interactive. Prompt for command line input, even if the standard input does not appear to be a terminal (character-special device).

–n Parse (interpret), but do not execute commands. This option can be used to check C shell scripts for syntax errors.

–s Take commands from the standard input.

–t Read and execute a single command line. A '\' (backslash) can be used to escape each newline for continuation of the command line onto subsequent input lines.

–v Verbose. Set the **verbose** predefined variable; command input is echoed after history substitution (but before other substitutions) and before execution.

–V Set **verbose** before reading **.cshrc**.

–x Echo. Set the **echo** variable; echo commands after all substitutions and just before execution.

–X Set **echo** before reading **.cshrc**.

Except with the options –**c**, –**i**, –**s**, or –**t**, the first nonoption *argument* is taken to be the name of a command or script. It is passed as argument zero, and subsequent arguments are added to the argument list for that command or script.

USAGE

Filename Completion

When enabled by setting the variable **filec**, an interactive C shell can complete a partially typed filename or user name. When an unambiguous partial filename is followed by an ESC character on the terminal input line, the shell fills in the remaining characters of a matching filename from the working directory.

If a partial filename is followed by the EOF character (usually typed as CTRL-d), the shell lists all filenames that match. It then prompts once again, supplying the incomplete command line typed in so far.

When the last (partial) word begins with a tilde (~), the shell attempts completion with a user name, rather than a file in the working directory.

The terminal bell signals errors or multiple matches; this can be inhibited by setting the variable **nobeep**. You can exclude files with certain suffixes by listing those suffixes in the variable **fignore**. If, however, the only possible completion includes a suffix in the list, it is not ignored. **fignore** does not affect the listing of filenames by the EOF character.

Lexical Structure

The shell splits input lines into words at space and tab characters, except as noted below. The characters **&**, **|**, **;**, **<**, **>**, **(**, and **)** form separate words; if paired, the pairs form single words. These shell metacharacters can be made part of other words, and their special meaning can be suppressed by preceding them with a '\' (backslash). A newline preceded by a \ is equivalent to a space character.

In addition, a string enclosed in matched pairs of single-quotes (´), double-quotes ("), or backquotes (` ), forms a partial word; metacharacters in such a string, including any space or tab characters, do not form separate words. Within pairs of backquote ( `) or double-quote (") characters, a newline preceded by a '\' (backslash) gives a true newline character. Additional functions of each type of quote are described, below, under **Variable Substitution**, **Command Substitution**, and **Filename Substitution**.

When the shell's input is not a terminal, the character **#** introduces a comment that continues to the end of the input line. Its special meaning is suppressed when preceded by a \ or enclosed in matching quotes.

Command Line Parsing

A *simple command* is composed of a sequence of words. The first word (that is not part of an I/O redirection) specifies the command to be executed. A simple command, or a set of simple commands separated by **|** or **|&** characters, forms a *pipeline*. With **|**, the standard output of the preceding command is redirected to the standard input of the command that follows. With **| &**, both the standard error and the standard output are redirected through the pipeline.

Pipelines can be separated by semicolons (**;**), in which case they are executed sequentially. Pipelines that are separated by **&&** or **| |** form conditional sequences in which the execution of pipelines on the right depends upon the success or failure, respectively, of the pipeline on the left.

A pipeline or sequence can be enclosed within parentheses '**( )**' to form a simple command that can be a component in a pipeline or sequence.

A sequence of pipelines can be executed asynchronously or "in the background" by appending an '**&**'; rather than waiting for the sequence to finish before issuing a prompt, the shell displays the job number (see **Job Control**, below) and associated process IDs and prompts immediately.

History Substitution

History substitution 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. Command lines are saved in the history list, the size of which is controlled by the **history** variable. The most recent command is retained in any case. A history substitution begins with a **!** (although you can change this with the **histchars** variable) and may occur anywhere on the command line; history substitutions do not nest. The **!** can be escaped with \ to suppress its special meaning.

Input lines containing history substitutions are echoed on the terminal after being expanded, but before any other substitutions take place or the command gets executed.

Event Designators

An event designator is a reference to a command line entry in the history list.

- **!** Start a history substitution, except when followed by a space character, tab, newline, **=** or **(**.
- **!!** Refer to the previous command. By itself, this substitution 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*.
!?*str***?** *additional*
Refer to the most recent command containing *str* and append *additional* to that referenced command.
!{*command***}** *additional*
Refer to the most recent command beginning with *command* and append *additional* to that referenced command.
^*previous_word***^***replacement***^**
Repeat the previous command line replacing the string *previous_word* with the string *replacement*. This is equivalent to the history substitution:
!:s/*previous_word***/***replacement***/**.

To re-execute a specific previous command AND make such a substitution, say, re-executing command #6,
!:6s/*previous_word***/***replacement***/**.

Word Designators

A '**:**' (colon) separates the event specification from the word designator. It can be omitted if the word designator begins with a **^**, **$**, *****, **–** or **%**. If the word is to be selected from the previous command, the second **!** character can be omitted from the event specification. For instance, **!!:1** and **!:1** both refer to the first word of the previous command, while **!!$** and **!$** both refer to the last word in the previous command. Word designators include:

The entire command line typed so far.
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) **?***s* search.
x–*y* A range of words; –*y* abbreviates **0**–*y*.
***** All the arguments, or a null value if there is just one word in the event.
*x****** Abbreviates *x*–**$**.
x– Like *x** but omitting word **$**.

Modifiers

After the optional word designator, you can add a sequence of one or more of the following modifiers, each preceded by a **:**.

h Remove a trailing pathname component, leaving the head.
r Remove a trailing suffix of the form '**.***xxx*', leaving the basename.
e Remove all but the suffix, leaving the Extension.
s/*l***/***r***/** Substitute *r* for *l*.
t Remove all leading pathname components, leaving the tail.
& Repeat the previous substitution.
g Apply the change to the first occurrence of a match in each word, by prefixing the above (for example, **g&**).
p Print the new command but do not execute it.
q Quote the substituted words, escaping further substitutions.

x Like **q**, but break into words at each space character, tab or newline.

Unless preceded by a **g**, the modification is applied only to the first string that matches *l*; an error results if no string matches.

The left-hand side of substitutions are not regular expressions, 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, is replaced by the text from the left-hand-side. The **&** can be quoted with a backslash. A null *l* uses the previous string either from a *l* or from a 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.

Without an event specification, a history reference refers either to the previous command, or to a previous history reference on the command line (if any).

Quick Substitution

^*l*^*r*^ This is equivalent to the history substitution: **!:s/***l***/***r***/**.

Aliases

The C shell maintains a list of aliases that you can create, display, and modify using the **alias** and **unalias** commands. The shell checks the first word in each command to see if it matches the name of an existing alias. If it does, the command is reprocessed with the alias definition replacing its name; the history substitution mechanism is made available as though that command were the previous input line. This allows history substitutions, escaped with a backslash in the definition, to be replaced with actual command line arguments when the alias is used. If no history substitution is called for, the arguments remain unchanged.

Aliases can be nested. That is, an alias definition can contain the name of another alias. Nested aliases are expanded before any history substitutions is applied. This is useful in pipelines such as

alias lm ´ls –l \!* | more´

which when called, pipes the output of **ls**(1) through **more**(1).

Except for the first word, the name of the alias may not appear in its definition, nor in any alias referred to by its definition. Such loops are detected, and cause an error message.

I/O Redirection

The following metacharacters indicate that the subsequent word is the name of a file to which the command's standard input, standard output, or standard error is redirected; this word is variable, command, and filename expanded separately from the rest of the command.

< Redirect the standard input.

<<*word* Read the standard input, up to a line that is identical with *word*, and place the resulting lines in a temporary file. Unless *word* is escaped or quoted, variable and command substitutions are performed on these lines. Then, the pipeline is invoked with the temporary file as its standard input. *word* is not subjected to variable, filename, or command substitution, and each line is compared to it before any substitutions are performed by the shell.

> >! >& >&! Redirect the standard output to a file. If the file does not exist, it is created. If it does exist, it is overwritten; its previous contents are lost.

When set, the variable **noclobber** prevents destruction of existing files. It also prevents redirection to terminals and **/dev/null**, unless one of the **!** forms is used. The **&** forms redirect both standard output and the standard error (diagnostic output) to the file.

>> >>& >>! >>&!
Append the standard output. Like **>**, but places output at the end of the file rather than overwriting it. If **noclobber** is set, it is an error for the file not to exist, unless one of the **!** forms is used. The **&** forms append both the standard error and standard output to the file.

Variable Substitution

The C shell maintains a set of *variables*, each of which is composed of a *name* and a *value*. A variable name consists of up to 20 letters and digits, and starts with a letter (the underscore is considered a letter). A variable's value is a space-separated list of zero or more words.

To refer to a variable's value, precede its name with a '**$**'. Certain references (described below) can be used to select specific words from the value, or to display other information about the variable. Braces can be used to insulate the reference from other characters in an input-line word.

Variable substitution takes place after the input line is analyzed, aliases are resolved, and I/O redirections are applied. Exceptions to this are variable references in I/O redirections (substituted at the time the redirection is made), and backquoted strings (see Command Substitution).

Variable substitution can be suppressed by preceding the **$** with a \, except within double-quotes where it always occurs. Variable substitution is suppressed inside of single-quotes. A **$** is escaped if followed by a space character, tab or newline.

Variables can be created, displayed, or destroyed using the **set** and **unset** commands. Some variables are maintained or used by the shell. For instance, the **argv** variable contains an image of the shell's argument list. Of the variables used by the shell, a number are toggles; the shell does not care what their value is, only whether they are set or not.

Numerical values can be operated on as numbers (as with the @ built-in command). With numeric operations, an empty value is considered to be zero; the second and subsequent words of multiword values are ignored. For instance, when the **verbose** variable is set to any value (including an empty value), command input is echoed on the terminal.

Command and filename substitution is subsequently applied to the words that result from the variable substitution, except when suppressed by double-quotes, when **noglob** is set (suppressing filename substitution), or when the reference is quoted with the **:q** modifier. Within double-quotes, a reference is expanded to form (a portion of) a quoted string; multiword values are expanded to a string with embedded space characters. When the **:q** modifier is applied to the reference, it is expanded to a list of space-separated words, each of which is quoted to prevent subsequent command or filename substitutions.

Except as noted below, it is an error to refer to a variable that is not set.

$*var*
${*var***}** These are replaced by words from the value of *var*, each separated by a space character. If *var* is an environment variable, its value is returned (but '**:**' modifiers and the other forms given below are not available).

$*var***[***index***]**
${***var***[***index***]}** These select only the indicated words from the value of *var*. Variable substitution is applied to *index* , which may consist of (or result in) a either single number, two numbers separated by a '–', or an asterisk. Words are indexed starting from 1; a '∗' selects all words. If the first number of a range is omitted (as with **$argv[–2]), it defaults to 1. If the last number of a range is omitted (as with **$argv[1–]**), it defaults to **$#***var* (the word count). It is not an error for a range to be empty if the second argument is omitted (or within range).

$#*name*
${#*name***}** These give the number of words in the variable.

$0 This substitutes the name of the file from which command input is being read except for setuid shell scripts. An error occurs if the name is not known.

$*n*
${***n***}** Equivalent to **$argv[*n***]**.

$∗** Equivalent to **$argv[∗].

The modifiers **:e**, **:h**, **:q**, **:r**, **:t**, and **:x** can be applied (see **History Substitution**), as can **:gh**, **:gt**, and **:gr**. If **{ }** (braces) are used, then the modifiers must appear within the braces. The current implementation allows only one such modifier per expansion.

The following references may not be modified with **:** modifiers.

$?*var*
${?*var***}** Substitutes the string 1 if *var* is set or 0 if it is not set.

$?0 Substitutes 1 if the current input filename is known or 0 if it is not.

$$ Substitute the process number of the (parent) shell.

$< Substitutes a line from the standard input, with no further interpretation thereafter. It can be used to read from the keyboard in a C shell script.

Command and Filename Substitutions

Command and filename substitutions are applied selectively to the arguments of built-in commands. Portions of expressions that are not evaluated are not expanded. For non-built-in commands, filename expansion of the command name is done separately from that of the argument list; expansion occurs in a subshell, after I/O redirection is performed.

Command Substitution

A command enclosed by backquotes (` . . . `) is performed by a subshell. Its standard output is broken into separate words at each space character, tab and newline; null words are discarded. This text replaces the backquoted string on the current command line.

Within double-quotes, only newline characters force new words; space and tab characters are preserved. However, a final newline is ignored. It is therefore possible for a command substitution to yield a partial word.

Filename Substitution

Unquoted words containing any of the characters **∗**, **?**, **[** or **{**, or that begin with **˜**, are expanded (also known as *globbing*) to an alphabetically sorted list of filenames, as follows:

∗ Match any (zero or more) characters.

? Match any single character.

[. . . **]** Match any single character in the enclosed list(s) or range(s). A list is a string of characters. A range is two characters separated by a dash (–), and includes all the characters in between in the ASCII collating sequence (see **ascii**(5)).

{ *str***,** *str***,** . . . **}**
Expand to each string (or filename-matching pattern) in the comma-separated list. Unlike the pattern-matching expressions above, the expansion of this construct is not sorted. For instance, **{b,a}** expands to '**b**' '**a**', (not '**a**' '**b**'). As special cases, the characters **{** and **}**, along with the string **{ }**, are passed undisturbed.

˜[*user*] Your home directory, as indicated by the value of the variable **home**, or that of *user*, as indicated by the password entry for *user*.

Only the patterns **∗**, **?** and **[**. . .**]** imply pattern matching; an error results if no filename matches a pattern that contains them. The '**.**' (dot character), when it is the first character in a filename or pathname component, must be matched explicitly. The **/** (slash) must also be matched explicitly.

Expressions and Operators

A number of C shell built-in commands accept expressions, in which the operators are similar to those of C and have the same precedence. These expressions typically appear in the **@**, **exit**, **if**, **set** and **while** commands, and are often used to regulate the flow of control for executing commands. Components of an expression are separated by white space.

Null or missing values are considered 0. The result of all expressions is a string, which may represent decimal numbers.

The following C shell operators are grouped in order of precedence:

(. . . **)**	grouping
˜	one's complement
!	logical negation
∗ / %	multiplication, division, remainder (These are right associative, which can lead to unexpected results. Group combinations explicitly with parentheses.)
+ −	addition, subtraction (also right associative)
<< >>	bitwise shift left, bitwise shift right
< > <= >=	less than, greater than, less than or equal to, greater than

	or equal to
== != =˜ !˜	equal to, not equal to, filename-substitution pattern match (described below), filename-substitution pattern mismatch
&	bitwise AND
ˆ	bitwise XOR (exclusive or)
|	bitwise inclusive OR
&&	logical AND
| |	logical OR

The operators: **==**, **!=**, **=˜**, and **!˜** compare their arguments as strings; other operators use numbers. The operators **=˜** and **!˜** each check whether or not a string to the left matches a filename substitution pattern on the right. This reduces the need for **switch** statements when pattern-matching between strings is all that is required.

Also available are file inquiries:

−**r** *filename*	Return true, or 1 if the user has read access. Otherwise it returns false, or 0.
−**w** *filename*	True if the user has write access.
−**x** *filename*	True if the user has execute permission (or search permission on a directory).
−**e** *filename*	True if *filename* exists.
−**o** *filename*	True if the user owns *filename*.
−**z** *filename*	True if *filename* is of zero length (empty).
−**f** *filename*	True if *filename* is a plain file.
−**d** *filename*	True if *filename* is a directory.

If *filename* does not exist or is inaccessible, then all inquiries return false.

An inquiry as to the success of a command is also available:

{ *command* **}**	If *command* runs successfully, the expression evaluates to true, 1. Otherwise, it evaluates to false, 0. (Note: Conversely, *command* itself typically returns 0 when it runs successfully, or some other value if it encounters a problem. If you want to get at the status directly, use the value of the **status** variable rather than this expression).

Control Flow

The shell contains a number of commands to regulate the flow of control in scripts and within limits, from the terminal. These commands operate by forcing the shell either to reread input (to *loop*), or to skip input under certain conditions (to *branch*).

Each occurrence of a **foreach**, **switch**, **while**, **if**. . .**then** and **else** built-in command must appear as the first word on its own input line.

If the shell's input is not seekable and a loop is being read, that input is buffered. The shell performs seeks within the internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward **goto** commands will succeed on nonseekable inputs.)

Command Execution

If the command is a C shell built-in command, the shell executes it directly. Otherwise, the shell searches for a file by that name with execute access. If the command name contains a **/**, the shell takes it as a pathname, and searches for it. If the command name does not contain a **/**, the shell attempts to resolve it to a pathname, searching each directory in the **path** variable for the command. To speed the search, the shell uses its hash table (see the **rehash** built-in command) to eliminate directories that have no applicable files. This hashing can be disabled with the −**c** or −**t**, options, or the **unhash** built-in command.

As a special case, if there is no **/** in the name of the script and there is an alias for the word **shell**, the expansion of the **shell** alias is prepended (without modification) to the command line. The system attempts to execute the first word of this special (late-occurring) alias, which should be a full pathname. Remaining words of the alias's definition, along with the text of the input line, are treated as arguments.

When a pathname is found that has proper execute permissions, the shell forks a new process and passes it, along with its arguments, to the kernel using the **execve**( ) system call (see **exec**(2)). The kernel then attempts to overlay the new process with the desired program. If the file is an executable binary (in **a.out**(4) format) the kernel succeeds and begins executing the new process. If the file is a text file and the first line begins with **#!**, the next word is taken to be the pathname of a shell (or command) to interpret that script. Subsequent words on the first line are taken as options for that shell. The kernel invokes (overlays) the indicated shell, using the name of the script as an argument.

If neither of the above conditions holds, the kernel cannot overlay the file and the **execve**( ) call fails (see **exec**(2)); the C shell then attempts to execute the file by spawning a new shell, as follows:

- If the first character of the file is a **#**, a C shell is invoked.
- Otherwise, a Bourne shell is invoked.

Signal Handling

The shell normally ignores QUIT signals. Background jobs are immune to signals generated from the keyboard, including hangups (HUP). Other signals have the values that the C shell inherited from its environment. The shell's handling of interrupt and terminate signals within scripts can be controlled by the **onintr** built-in command. Login shells catch the TERM signal; otherwise, this signal is passed on to child processes. In no case are interrupts allowed when a login shell is reading the **.logout** file.

Job Control

The shell associates a numbered *job* with each command sequence to keep track of those commands that are running in the background or have been stopped with TSTP signals (typically CTRL-z). When a command or command sequence (semicolon separated list) is started in the background using the **&** metacharacter, the shell displays a line with the job number in brackets and a list of associated process numbers:

[1] 1234

To see the current list of jobs, use the **jobs** built-in command. The job most recently stopped (or put into the background if none are stopped) is referred to as the *current* job and is indicated with a '**+**'. The previous job is indicated with a '−'; when the current job is terminated or moved to the foreground, this job takes its place (becomes the new current job).

To manipulate jobs, refer to the **bg**, **fg**, **kill**, **stop**, and % built-in commands.

A reference to a job begins with a '%'. By itself, the percent-sign refers to the current job.

% %+ %%	The current job.
%−	The previous job.
%*j*	Refer to job *j* as in: '**kill −9** %*j*'. *j* can be a job number, or a string that uniquely specifies the command line by which it was started; '**fg %vi**' might bring a stopped **vi** job to the foreground, for instance.
%**?***string*	Specify the job for which the command line uniquely contains *string*.

A job running in the background stops when it attempts to read from the terminal. Background jobs can normally produce output, but this can be suppressed using the '**stty tostop**' command.

Status Reporting

While running interactively, the shell tracks the status of each job and reports whenever the job finishes or becomes blocked. It normally displays a message to this effect as it issues a prompt, in order to avoid disturbing the appearance of your input. When set, the **notify** variable indicates that the shell is to report status changes immediately. By default, the **notify** command marks the current process; after starting a background job, type **notify** to mark it.

Built-In Commands

Built-in commands are executed within the C shell. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell.

: Null command. This command is interpreted, but performs no action.

alias [*name* [*def*]]
Assign *def* to the alias *name*. *def* is a list of words that may contain escaped history-substitution metasyntax. *name* is not allowed to be **alias** or **unalias**. If *def* is omitted, the alias *name* is displayed along with its current definition. If both *name* and *def* are omitted, all aliases are displayed.

bg [%*job* . . .]
Run the current or specified jobs in the background.

break Resume execution after the **end** of the nearest enclosing **foreach** or **while** loop. The remaining commands on the current line are executed. This allows multilevel breaks to be written as a list of **break** commands, all on one line.

breaksw Break from a **switch**, resuming after the **endsw**.

case *label***:** A label in a **switch** statement.

cd [*dir*]
chdir [*dir*]
Change the shell's working directory to directory *dir*. If no argument is given, change to the home directory of the user. If *dir* is a relative pathname not found in the current directory, check for it in those directories listed in the **cdpath** variable. If *dir* is the name of a shell variable whose value starts with a **/**, change to the directory named by that value.

continue Continue execution of the next iteration of the nearest enclosing **while** or

foreach loop.

default: Labels the default case in a **switch** statement. The default should come after all **case** labels. Any remaining commands on the command line are first executed.

dirs [–**l**]
Print the directory stack, most recent to the left; the first directory shown is the current directory. With the –**l** argument, produce an unabbreviated printout; use of the ˜ notation is suppressed.

echo [–**n**] *list*
The words in *list* are written to the shell's standard output, separated by space characters. The output is terminated with a newline unless the –**n** option is used.
csh will, by default, invoke its built-in **echo**, if **echo** is called without the full pathname of a Unix command, regardless of the configuration of your **PATH** (see **echo**(1)).

eval *argument* . . .
Reads the arguments as input to the shell and executes the resulting command(s). This is usually used to execute commands generated as the result of command or variable substitution. See **tset**(1B) for an example of how to use **eval**.

exec *command*
Execute *command* in place of the current shell, which terminates.

exit [**(***expr***)**]
The calling shell or shell script exits, either with the value of the status variable or with the value specified by the expression *expr* .

fg [%*job*]
Bring the current or specified *job* into the foreground.

foreach *var* **(***wordlist***)**
. . .
end The variable *var* is successively set to each member of *wordlist*. The sequence of commands between this command and the matching **end** is executed for each new value of *var*. Both **foreach** and **end** must appear alone on separate lines.

The built-in command **continue** may be used to terminate the execution of the current iteration of the loop and the built-in command **break** may be used to terminate execution of the **foreach** command. When this command is read from the terminal, the loop is read once prompting with **?** before any statements in the loop are executed.

glob *wordlist*
Perform filename expansion on *wordlist*. Like **echo**, but no **** escapes are recognized. Words are delimited by **NULL** characters in the output.

goto *label* The specified *label* is a filename and a command expanded to yield a label.

The shell rewinds its input as much as possible and searches for a line of the form *label***:** possibly preceded by space or tab characters. Execution continues after the indicated line. It is an error to jump to a label that occurs between a **while** or **for** built-in command and its corresponding **end**.

hashstat Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding **exec**s). An **exec** is attempted for each component of the *path* where the hash function indicates a possible hit and in each component that does not begin with a '/'.

history [**–hr**] [*n*]
Display the history list; if *n* is given, display only the *n* most recent events.

- **–r** Reverse the order of printout to be most recent first rather than oldest first.
- **–h** Display the history list without leading numbers. This is used to produce files suitable for sourcing using the **–h** option to *source*.

if (*expr***)** *command*
If the specified expression evaluates to true, the single *command* with arguments is executed. Variable substitution on *command* happens early, at the same time it does for the rest of the *if* command. *command* must be a simple command, not a pipeline, a command list, or a parenthesized command list. Note: I/O redirection occurs even if *expr* is false, when *command* is *not* executed (this is a bug).

if (*expr***) then**
. . .
else if (*expr2***) then**
. . .
else
. . .
endif If *expr* is true, commands up to the first **else** are executed. Otherwise, if *expr2* is true, the commands between the **else if** and the second **else** are executed. Otherwise, commands between the **else** and the **endif** are executed. Any number of **else if** pairs are allowed, but only one **else**. Only one **endif** is needed, but it is required. The words **else** and **endif** must be the first nonwhite characters on a line. The **if** must appear alone on its input line or after an **else**.

jobs[**–l**] List the active jobs under job control.

- **–l** List process IDs, in addition to the normal information.

kill [*–sig*] [*pid*] [%*job*] . . .
kill –l Send the **TERM** (terminate) signal, by default, or the signal specified, to the specified process ID, the *job* indicated, or the current *job*. Signals are either given by number or by name. There is no default. Typing **kill** does not send a signal to the current job. If the signal being sent is **TERM** (terminate) or **HUP** (hangup), then the job or process is sent a **CONT** (continue) signal as well.

−l List the signal names that can be sent.

limit [−**h**] [*resource* [*max-use*]]

Limit the consumption by the current process or any process it spawns, each not to exceed *max-use* on the specified *resource*. If *max-use* is omitted, print the current limit; if *resource* is omitted, display all limits. (Run the **sysdef**(1M) command to obtain the maximum possible limits for your system. The values reported are in hexidecimal, but can be translated into decimal numbers using the **bc**(1) command).

−h Use hard limits instead of the current limits. Hard limits impose a ceiling on the values of the current limits. Only the privileged user may raise the hard limits.

resource is one of:

cputime Maximum CPU seconds per process.
filesize Largest single file allowed; limited to the size of the filesystem. (see **df**(1M)).
datasize (heapsize)
Maximum data size (including stack) for the process. This is the size of your virtual memory (see **swap**(1M)).
stacksize Maximum stack size for the process. (see **swap**(1M)).
coredumpsize Maximum size of a core dump (file). This limited to the size of the filesystem.
descriptors Maximum number of file descriptors. (run **sysdef( ).**
memorysize Maximum size of virtual memory.

max-use is a number, with an optional scaling factor, as follows:

*n***h** Hours (for **cputime**).
*n***k** *n* kilobytes. This is the default for all but **cputime**.
*n***m** *n* megabytes or minutes (for **cputime**).
*mm***:***ss* Minutes and seconds (for **cputime**).

Example of limit: to limit the size of a core file dump to 0 Megabytes, type the following:

limit coredumpsize 0M

login [*username* | −**p**]

Terminate a login shell and invoke **login**(1). The **.logout** file is not processed. If *username* is omitted, **login** prompts for the name of a user.

−p Preserve the current environment (variables).

logout Terminate a login shell.

nice [**+***n* | −*n*] [*command*]

Increment the process priority value for the shell or for *command* by *n*. The higher the priority value, the lower the priority of a process, and the slower it runs. When given, *command* is always run in a subshell, and the restrictions placed on commands in simple **if** commands apply. If *command* is omitted, **nice** increments the value for the current shell. If no increment is specified, **nice** sets the process priority value to 4. The range of process priority values is from −20 to 20. Values of *n* outside this range set the value to the lower, or to the higher boundary, respectively.

+*n* Increment the process priority value by *n*.

−*n* Decrement by *n*. This argument can be used only by the privileged user.

nohup [*command*]
Run *command* with HUPs ignored. With no arguments, ignore HUPs throughout the remainder of a script. When given, *command* is always run in a subshell, and the restrictions placed on commands in simple **if** statements apply. All processes detached with **&** are effectively **nohup**'d.

notify [%*job*] . . .
Notify the user asynchronously when the status of the current job or specified jobs changes.

onintr [−| *label*]
Control the action of the shell on interrupts. With no arguments, **onintr** restores the default action of the shell on interrupts. (The shell terminates shell scripts and returns to the terminal command input level). With the − argument, the shell ignores all interrupts. With a *label* argument, the shell executes a **goto** *label* when an interrupt is received or a child process terminates because it was interrupted.

popd [**+***n*] Pop the directory stack and **cd** to the new top directory. The elements of the directory stack are numbered from 0 starting at the top.

+*n* Discard the *n*'th entry in the stack.

pushd [**+***n* | *dir*]
Push a directory onto the directory stack. With no arguments, exchange the top two elements.

+*n* Rotate the *n*'th entry to the top of the stack and **cd** to it.

dir Push the current working directory onto the stack and change to *dir*.

rehash Recompute the internal hash table of the contents of directories listed in the *path* variable to account for new commands added.

repeat *count command*
Repeat *command count* times. *command* is subject to the same restrictions as with the one-line **if** statement.

set [*var* [= *value*]]
set *var*[*n*] = *word*
With no arguments, **set** displays the values of all shell variables. Multiword values are displayed as a parenthesized list. With the *var* argument alone, **set** assigns an empty (null) value to the variable *var*. With arguments of the form

var **=** *value* **set** assigns *value* to *var*, where *value* is one of:

word A single word (or quoted string).
(*wordlist***)** A space-separated list of words enclosed in parentheses.

Values are command and filename expanded before being assigned. The form **set** *var***[***n***]** **=** *word* replaces the *n*'th word in a multiword value with *word*.

setenv [*VAR* [*word*]]

With no arguments, **setenv** displays all environment variables. With the *VAR* argument, **setenv** sets the environment variable *VAR* to have an empty (null) value. (By convention, environment variables are normally given upper-case names.) With both *VAR* and *word* arguments, **setenv** sets the environment variable **NAME** to the value *word*, which must be either a single word or a quoted string. The most commonly used environment variables, **USER**, **TERM**, and **PATH**, are automatically imported to and exported from the **csh** variables **user**, **term**, and **path**; there is no need to use **setenv** for these. In addition, the shell sets the **PWD** environment variable from the **csh** variable **cwd** whenever the latter changes.

The environment variables **LC_CTYPE, LC_MESSAGES, LC_TIME, LC_COLLATE, LC_NUMERIC,** and **LC_MONETARY** take immediate effect when changed within the C shell.

If any of the **LC_*** variables (**LC_CTYPE, LC_MESSAGES, LC_TIME, LC_COLLATE, LC_NUMERIC,** and **LC_MONETARY**) (see **environ**(5)) are not set in the environment, the operational behavior of **csh** for each corresponding locale category is determined by the value of the **LANG** environment variable. If **LC_ALL** is set, its contents are used to override both the **LANG** and the other **LC_*** variables. If none of the above variables is set in the environment, the "C" (U.S. style) locale determines how **csh** behaves.

LC_CTYPE

Determines how **csh** handles characters. When **LC_CTYPE** is set to a valid value, **csh** can display and handle text and filenames containing valid characters for that locale. **csh** can display and handle Extended Unix Code (EUC) characters where any individual character can be 1, 2, or 3 bytes wide. **csh** can also handle EUC characters of 1, 2, or more column widths. In the "C" locale, only characters from ISO 8859-1 are valid.

LC_MESSAGES

Determines how diagnostic and informative messages are presented. This includes the language and style of the messages and the correct form of affirmative and negative responses. In the "C" locale, the messages are presented in the default form found in the program itself (in most cases, U.S./English).

LC_NUMERIC
Determines the value of the radix character (decimal point (".") in the "C" locale) and thousand separator (empty string ("") in the "C" locale).

shift [*variable*]
The components of **argv**, or *variable*, if supplied, are shifted to the left, discarding the first component. It is an error for the variable not to be set or to have a null value.

source [−**h**] *name*
Reads commands from *name*. **source** commands may be nested, but if they are nested too deeply the shell may run out of file descriptors. An error in a sourced file at any level terminates all nested **source** commands.

−**h** Place commands from the file *name* on the history list without executing them.

stop %*jobid* . . .
Stop the current or specified background job.

stop *pid* . . .
Stop the specified process, *pid*. (see **ps**(1)).

suspend Stop the shell in its tracks, much as if it had been sent a stop signal with ^**Z**. This is most often used to stop shells started by **su**.

switch (*string***)**
case *label***:**
. . .
breaksw
. . .
default:
. . .
breaksw
endsw
Each *label* is successively matched, against the specified *string*, which is first command and filename expanded. The file metacharacters *****, **?** and **[**. . .**]** may be used in the case labels, which are variable expanded. If none of the labels match before a “default” label is found, execution begins after the default label. Each **case** statement and the **default** statement must appear at the beginning of a line. The command **breaksw** continues execution after the **endsw**. Otherwise control falls through subsequent **case** and **default** statements as with C. If no label matches and there is no default, execution continues after the **endsw**.

time [*command*]
With no argument, print a summary of time used by this C shell and its children. With an optional *command*, execute *command* and print a summary of the time it uses.

As of this writing, the **time** built-in command does NOT compute the last 6 fields of output, rendering the output to erroneously report the value "**0**" for these fields.

example %time ls -R
9.0u 11.0s 3:32 10%

(See below the "Environment Variables and Predefined Shell Variables" subsection on the **time** variable.)

umask [*value*]
Display the file creation mask. With *value*, set the file creation mask. With *value* given in octal, the user can turn-off any bits, but cannot turn-on bits to allow new permissions. Common values include 077, restricting all permissions from everyone else; 002, giving complete access to the group, and read (and directory search) access to others; or 022, giving read (and directory search) but not write permission to the group and others.

unalias *pattern*
Discard aliases that match (filename substitution) *pattern*. All aliases are removed by '**unalias** *'.

unhash Disable the internal hash table.

unlimit [–**h**] [*resource*]
Remove a limitation on *resource*. If no *resource* is specified, then all resource limitations are removed. See the description of the **limit** command for the list of resource names.

–**h** Remove corresponding hard limits. Only the privileged user may do this.

unset *pattern*
Remove variables whose names match (filename substitution) *pattern*. All variables are removed by '**unset** *'; this has noticeably distasteful side effects.

unsetenv *variable*
Remove *variable* from the environment. As with **unset**, pattern matching is not performed.

wait Wait for background jobs to finish (or for an interrupt) before prompting.

while (*expr***)**
. . .
end While *expr* is true (evaluates to nonzero), repeat commands between the **while** and the matching **end** statement. **break** and **continue** may be used to terminate or continue the loop prematurely. The **while** and **end** must appear alone on their input lines. If the shell's input is a terminal, it prompts for commands with a question-mark until the **end** command is entered and then performs the commands in the loop.

%[*job*] [**&**]
Bring the current or indicated *job* to the foreground. With the ampersand, continue running *job* in the background.

@ [*var* **=***expr*]
@ [*var***[***n***]** **=***expr*]
With no arguments, display the values for all shell variables. With arguments, set the variable *var*, or the *n*'th word in the value of *var*, to the value that *expr* evaluates to. (If **[***n***]** is supplied, both *var* and its *n*'th component must already exist.)

If the expression contains the characters **>**, **<**, **&**, or **|**, then at least this part of *expr* must be placed within parentheses.

The operators ***=**, **+=**, and so forth, are available as in C. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* that would otherwise be single words.

Special postfix operators, **+ +** and **– –**, increment or decrement *name*, respectively.

Environment Variables and Predefined Shell Variables

Unlike the Bourne shell, the C shell maintains a distinction between environment variables, which are automatically exported to processes it invokes, and shell variables, which are not. Both types of variables are treated similarly under variable substitution. The shell sets the variables **argv**, **cwd**, **home**, **path**, **prompt**, **shell**, and **status** upon initialization. The shell copies the environment variable **USER** into the shell variable **user**, **TERM** into **term**, and **HOME** into **home**, and copies each back into the respective environment variable whenever the shell variables are reset. **PATH** and **path** are similarly handled. You need only set **path** once in the **.cshrc** or **.login** file. The environment variable **PWD** is set from **cwd** whenever the latter changes. The following shell variables have predefined meanings:

argv Argument list. Contains the list of command line arguments supplied to the current invocation of the shell. This variable determines the value of the positional parameters **$1**, **$2**, and so on.

cdpath Contains a list of directories to be searched by the **cd**, **chdir**, and **popd** commands, if the directory argument each accepts is not a subdirectory of the current directory.

cwd The full pathname of the current directory.

echo Echo commands (after substitutions) just before execution.

fignore A list of filename suffixes to ignore when attempting filename completion. Typically the single word '**.o**'.

filec Enable filename completion, in which case the CTRL-d character EOT and the ESC character have special significance when typed in at the end of a terminal input line:

EOT Print a list of all filenames that start with the preceding string.

ESC Replace the preceding string with the longest unambiguous extension.

hardpaths If set, pathnames in the directory stack are resolved to contain no symbolic-link components.

histchars A two-character string. The first character replaces **!** as the history-substitution character. The second replaces the carat (^) for quick substitutions.

history The number of lines saved in the history list. A very large number may use up all of the C shell's memory. If not set, the C shell saves only the most recent command.

home The user's home directory. The filename expansion of ~ refers to the value of this variable.

ignoreeof If set, the shell ignores EOF from terminals. This protects against accidentally killing a C shell by typing a CTRL-d.

mail A list of files where the C shell checks for mail. If the first word of the value is a number, it specifies a mail checking interval in seconds (default 5 minutes).

nobeep Suppress the bell during command completion when asking the C shell to extend an ambiguous filename.

noclobber Restrict output redirection so that existing files are not destroyed by accident. **>** redirections can only be made to new files. **>>** redirections can only be made to existing files.

noglob Inhibit filename substitution. This is most useful in shell scripts once filenames (if any) are obtained and no further expansion is desired.

nonomatch Returns the filename substitution pattern, rather than an error, if the pattern is not matched. Malformed patterns still result in errors.

notify If set, the shell notifies you immediately as jobs are completed, rather than waiting until just before issuing a prompt.

path The list of directories in which to search for commands. **path** is initialized from the environment variable **PATH**, which the C shell updates whenever **path** changes. A null word specifies the current directory. The default is typically **(/usr/bin .)**. If **path** becomes unset only full pathnames will execute. An interactive C shell will normally hash the contents of the directories listed after reading **.cshrc**, and whenever **path** is reset. If new commands are added, use the **rehash** command to update the table.

prompt The string an interactive C shell prompts with. Noninteractive shells leave the **prompt** variable unset. Aliases and other commands in the **.cshrc** file that are only useful interactively, can be placed after the following test: '**if ($?prompt == 0) exit**', to reduce startup time for noninteractive shells. A **!** in the **prompt** string is replaced by the current event number. The default prompt is *hostname***%** for mere mortals, or *hostname***#** for the privileged user.

The setting of **$prompt** has three meanings:

$prompt not set -- non-interactive shell, test **$?prompt**.

$prompt set but == "" -- **.cshrc** called by the **which**(1) command.

$prompt set and != "" -- normal interactive shell.

savehist The number of lines from the history list that are saved in **~/.history** when the user logs out. Large values for **savehist** slow down the C shell during startup.

shell The file in which the C shell resides. This is used in forking shells to interpret files that have execute bits set, but that are not executable by the system.

status The status returned by the most recent command. If that command terminated abnormally, 0200 is added to the status. Built-in commands that fail return exit status 1; all other built-in commands set status to 0.

time Control automatic timing of commands. Can be supplied with one or two values. The first is the reporting threshold in CPU seconds. The second is a string of tags and text indicating which resources to report on. A tag is a percent sign (**%**) followed by a single upper-case letter (unrecognized tags print as text):

- **%D** Average amount of unshared data space used in Kilobytes.
- **%E** Elapsed (wallclock) time for the command.
- **%F** Page faults.
- **%I** Number of block input operations.
- **%K** Average amount of unshared stack space used in Kilobytes.
- **%M** Maximum real memory used during execution of the process.
- **%O** Number of block output operations.
- **%P** Total CPU time — U (user) plus S (system) — as a percentage of E (elapsed) time.
- **%S** Number of seconds of CPU time consumed by the kernel on behalf of the user's process.
- **%U** Number of seconds of CPU time devoted to the user's process.
- **%W** Number of swaps.

%X Average amount of shared memory used in Kilobytes.

The default summary display outputs from the **%U**, **%S**, **%E**, **%P**, **%X**, **%D**, **%I**, **%O**, **%F**, and **%W** tags, in that order.

verbose Display each command after history substitution takes place.

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shells after **.cshrc** at login.
~/.logout	Read by login shells at logout.
~/.history	Saved history for use at next login.
/usr/bin/sh	The Bourne shell, for shell scripts not starting with a '**#**'.
/tmp/sh*	Temporary file for '**<<**'.
/etc/passwd	Source of home directories for '*~name*'.

SEE ALSO

bc(1), **echo**(1), **login**(1), **ps**(1), **sh**(1), **shell_builtins**(1), **which**(1), **tset**(1B), **df**(1M), **swap**(1M), **sysdef**(1M), **access**(2), **exec**(2), **fork**(2), **pipe**(2), **a.out**(4), **environ**(4), **environ**(5), **ascii**(5), **termio**(7I)

DIAGNOSTICS

You have stopped jobs.
You attempted to exit the C shell with stopped jobs under job control. An immediate second attempt to exit will succeed, terminating the stopped jobs.

NOTES

Words can be no longer than 1024 characters. The system limits argument lists to 1,048,576 characters. However, the maximum number of arguments to a command for which filename expansion applies is 1706. Command substitutions may expand to no more characters than are allowed in the argument list. To detect looping, the shell restricts the number of **alias** substitutions on a single line to 20.

When a command is restarted from a stop, the shell prints the directory it started in if this is different from the current directory; this can be misleading (that is, wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/restartable. Command sequences of the form *a* **;** *b* **;** *c* are also not handled gracefully when stopping is attempted. If you suspend *b*, the shell never executes *c*. This is especially noticeable if the expansion results from an alias. It can be avoided by placing the sequence in parentheses to force it into a subshell.

Control over terminal output after processes are started is primitive; use the Sun Window system if you need better output control.

Commands within loops, prompted for by **?**, are not placed in the *history* list.

Control structures should be parsed rather than being recognized as built-in commands. This would allow control commands to be placed anywhere, to be combined with **|**, and to be used with **&** and **;** metasyntax.

It should be possible to use the **:** modifiers on the output of command substitutions. There are two problems with **:** modifier usage on variable substitutions: not all of the modifiers are available, and only one modifier per substitution is allowed.

The **g** (global) flag in history substitutions applies only to the first match in each word, rather than all matches in all words. The common text editors consistently do the latter when given the **g** flag in a substitution command.

Quoting conventions are confusing. Overriding the escape character to force variable substitutions within double quotes is counterintuitive and inconsistent with the Bourne shell.

Symbolic links can fool the shell. Setting the **hardpaths** variable alleviates this.

It is up to the user to manually remove all duplicate pathnames accrued from using built-in commands as

set path = *pathnames*

or

setenv PATH *pathnames*

more than once. These often occur because a shell script or a **.cshrc** file does something like '**set path=(/usr/local /usr/hosts $path)**' to ensure that the named directories are in the pathname list.

The only way to direct the standard output and standard error separately is by invoking a subshell, as follows:

example% (*command* **>** *outfile* **) >&** *errorfile*

Although robust enough for general use, adventures into the esoteric periphery of the C shell may reveal unexpected quirks.

If you start **csh** as a login shell and you do not have a **.login** in your home directory, then the **csh** reads in the **/etc/.login**.

BUGS

As of this writing, the **time** built-in command does NOT compute the last 6 fields of output, rendering the output to erroneously report the value "**0**" for these fields.

```
example %time ls -R
9.0u 11.0s 3:32 10% 0+0k 0+0io 0pf+0w
```