NAME

csh − a shell (command interpreter) with C-like syntax

SYNOPSIS

csh [-cefinstvxTVX] [*command_file*] [*argument_list* ...]

DESCRIPTION

csh is a command language interpreter that incorporates a command history buffer, C-like syntax, and job control facilities.

Command Options

Command options are interpreted as follows:

-c Read commands from the (single) following argument which must be present. Any remaining arguments are placed in

-e C shell exits if any invoked command terminates abnormally or yields a non-zero exit status.

-f Suppress execution of the **.cshrc** file in your home directory, thus speeding up shell start-up time.

-i Force **csh** to respond interactively when called from a device other than a computer terminal (such as another computer). **csh** normally responds non-interactively. If **csh** is called from a computer terminal, it always responds interactively, regardless of which options are selected.

-n Parse but do *not* execute commands. This is useful for checking syntax in shell scripts. All substitutions are performed (history, command, alias, etc.).

-s Take command input from the standard input.

-t Read and execute a single line of input.

-v Set the **verbose** shell variable, causing command input to be echoed to the standard output device after history substitutions are made.

-x Set the **echo** shell variable, causing all commands to be echoed to the standard error immediately before execution.

-T Disable the tenex features which use the ESC key for command/file name completion and **CTRL- D** for listing available files (see the *CSH UTILITIES* section below)

-V Set the **verbose** variable before **.cshrc** is executed so that all **.cshrc** commands are also echoed to the standard output.

-X Set the **echo** variable before **.cshrc** is executed so that all **.cshrc** commands are also echoed to the standard output.

After processing the command options, if arguments remain in the argument list, and the or **-t** options were not specified, the first remaining argument is taken as the name of a file of commands to be executed.

COMMANDS

A simple command is a sequence of words, the first of which specifies the command to be executed. A sequence of simple commands separated by vertical bar (**|**) characters forms a pipeline. The output of each command in a pipeline becomes the input for the next command in the pipeline. Sequences of pipelines can be separated by semicolons **(;)** which causes them to be executed sequentially. A sequence of pipelines can be executed in background mode by adding an ampersand character **(&)** after the last entry.

Any pipeline can be placed in parentheses to form a simple command which, in turn, can be a component of another pipeline. Pipelines can also be separated by **||** or **&&** indicating, as in the C language, that the second pipeline is to be executed only if the first fails or succeeds, respectively.

Jobs

csh associates a **job** with each pipeline and keeps a table of current jobs (printed by the **jobs** command) and assigns them small integer numbers. When a job is started asynchronously using the shell prints a line resembling:

[1] 1234

indicating that the job which was started asynchronously was job number 1 and had one (top-level) process, whose process id was 1234.

If you are running a job and want to do something else, you can type the currently defined *suspend* character (see *termio*(7)) which sends a stop signal to the current job. **csh** then normally indicates that the job has been 'Stopped', and prints another prompt. You can then manipulate the state of this job, putting it in the background with the **bg** command, run some other commands, and then eventually bring the job back into the foreground with the foreground command A *suspend* takes effect immediately and is like an interrupt in that pending output and unread input are discarded when it is typed. There is a delayed *suspend* character which does not generate a stop signal until a program attempts to *read*(2) it. This can usefully be typed ahead when you have prepared some commands for a job which you want to stop after it has read them.

A job being run in the background stops if it tries to read from the terminal. Background jobs are normally allowed to produce output, but this can be disabled by giving the command **stty tostop** (see *stty*(1)). If you set this tty option, background jobs stop when they try to produce output, just as they do when they try to read input. Keyboard signals and line-hangup signals from the terminal interface are not sent to background jobs on such systems. This means that background jobs are immune to the effects of logging out or typing the interrupt, quit, suspend, and delayed suspend characters (see *termio*(7)).

There are several ways to refer to jobs in the shell. The character **%** introduces a job name. If you wish to refer to job number 1, you can name it as Just naming a job brings it to the foreground; thus **%1** is a synonym for **fg %1 ,** bringing job 1 back into the foreground. Similarly, typing **%1 &** resumes job 1 in the background. Jobs can also be named by prefixes of the string typed in to start them if these prefixes are unambiguous; thus **%ex** normally restarts a suspended *ex*(1) job, if there is only one suspended job whose name begins with the string It is also possible to say **%?string** which specifies a job whose text contains *string*, if there is only one such job.

csh maintains a notion of the current and previous jobs. In output pertaining to jobs, the current job is marked with a **+** and the previous job with a The abbreviation **%+** refers to the current job and **%-** refers to the previous job. For close analogy with the syntax of the **history** mechanism (described below), **%%** is also a synonym for the current job.

csh learns immediately whenever a process changes state. It normally informs you whenever a job becomes blocked so that no further progress is possible, but only just before printing a prompt. This is done so that it does not otherwise disturb your work. If, however, you set the shell variable **csh** notifies you immediately of changes in status of background jobs. There is also a **csh** built-in command called **notify** which marks a single process so that any status change is immediately reported. By default, **notify** marks the current process. Simply type **notify** after starting a background job to mark it.

If you try to leave the shell while jobs are stopped, **csh** sends the warning message: **You have stopped jobs.** Use the **jobs** command to see what they are. If you do this or immediately try to exit again, **csh** does not warn you a second time, and the suspended jobs are terminated (see *exit*(2)).

Built-In Commands

Built-in commands are executed within the shell without spawning a new process. If a built-in command occurs as any component of a pipeline except the last, it is executed in a subshell. The built-in commands are:

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for *name*. The third form assigns the specified *wordlist* as the alias of *name*. Command and file name substitution are performed on *wordlist*. *name* cannot be **alias** or

Put the current
(*job* not specified) or specified jobs into the background, continuing them if they were stopped.

break Causes execution to resume after the **end** of the nearest enclosing **foreach** or The remaining commands on the current line are executed. Multi-level breaks are thus possible by writing them all on one line.

breaksw
Causes a break from a resuming after the

case label :
A label in a **switch** statement as discussed below.

cd
cd directory_name
chdir
chdir directory_name
Change the shell's current working directory to *directory_name*. If not specified, *directory_name* defaults to your home directory.
If *directory_name* is not found as a subdirectory of the current working directory (and does not begin with or each component of the variable *cdpath* is checked to see if it has a subdirectory *directory_name*. Finally, if all else fails, **csh** treats *directory_name* as a shell variable. If its value begins with this is tried to see if it is a directory.

continue
Continue execution of the nearest enclosing **while** or The rest of the commands on the current line are executed.

default: Labels the default case in a **switch** statement. The default should come after all other **case** labels.

dirs Prints the directory stack; the top of the stack is at the left; the first directory in the stack is the current directory.

echo wordlist
echo -n wordlist
The specified words are written to the shell's standard output, separated by spaces, and terminated with a new-line unless the **-n** option is specified.

else
end
endif
endsw See the descriptions of the and **while** statements below.

eval arguments ...
(Same behavior as *sh*(1).) *arguments* are read as input to the shell and the resulting command(s) executed. This is usually used to execute commands generated as the result of command or variable substitution, since parsing occurs before these substitutions.

exec command
The specified command is executed in place of the current shell.

exit
exit (expression)
csh exits either with the value of the **status** variable (first form) or with the value of the specified *expression* (second form).

Brings the current
(*job* not specified) or specified jobs into the foreground, continuing them if they were stopped.

foreach name (wordlist)

. . .

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

The built-in command **continue** can be used to continue the loop prematurely; the built-in command **break** to terminate it prematurely. When this command is read from the terminal, the loop is read once, prompting with **?** before any statements in the loop are executed. If you make a mistake while typing in a loop at the terminal, use the erase or line-kill character as appropriate to recover.

glob wordlist

Like **echo** but no **** escapes are recognized and words are delimited by null characters in the output. Useful in programs that use the shell to perform file name expansion on a list of words.

goto word

The specified *word* is file name and command expanded to yield a string of the form The shell rewinds its input as much as possible and searches for a line of the form **label:** possibly preceded by blanks or tabs. Execution continues after the specified line.

hashstat

Print a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding 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

Displays the history event list.

If *n* is given, only the *n* most recent events are printed. The **-r** option reverses the order of printout to be most recent first rather than oldest first. The **-h** option prints the history list without leading numbers for producing files suitable for the **source** command.

if (expression) command

If *expression* evaluates 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, a parenthesized command list, or an aliased command. Input/output redirection occurs even if *expression* is false, meaning that *command* is *not* executed (this is a bug).

if (expression1) then

. . .

else if (expression2) then

. . .

else

. . .

endif If *expression1* is true, all commands down to the first **else** are executed; otherwise if *expression2* is true, all commands from the first **else** down to the second **else** are executed, etc. Any number of pairs are possible, but only one **endif** is needed. The **else** part is likewise optional. (The words **else** and **endif** must appear at the beginning of input lines. The **if** must appear alone on its input line or after an

Lists active jobs.

The **-l** option lists process IDs in addition to the usual information.

kill % job
kill - sig % job ...
kill pid

kill - sig pid ...

kill -l Sends either the TERM (terminate) signal or the specified signal to the specified jobs or processes. Signals are either given by number or by names (as given in stripped of the **SIG** prefix (see *signal*(2)). The signal names are listed by There is no default, so **kill** used alone does not send a signal to the current job. If the signal being sent is TERM (terminate) or HUP (hangup), the job or process is sent a CONT (continue) signal as well.

login Terminates a login shell, replacing it with an instance of This is one way to log off, included for compatibility with *sh*(1).

logout Terminates a login shell. Especially useful if *ignoreeof* is set. A similar function, which works for sessions that are not login shells, is provided for historical reasons. Its use is not recommended because it is not part of the standard BSD **csh** and may not be supported in future releases.

newgrp Changes the group identification of the caller; for details see *newgrp*(1). A new shell is executed by **newgrp** so that the current shell environment is lost.

nice
nice + *number*
nice command
nice + *number command*

The first form sets the *nice* (run command priority) for this shell to 4 (the default). The second form sets the priority to the given *number*. The final two forms run *command* at priority 4 and *number* respectively. The user with appropriate privileges can raise the priority by specifying negative niceness using *nice - number ... command* is always executed in a sub-shell, and restrictions placed on commands in simple **if** statements apply.

Without an argument,

nohup can be used in shell scripts to cause hangups to be ignored for the remainder of the script. With an argument, causes the specified *command* to be run with hangups ignored. All processes executed in the background with **&** are effectively as described under Jobs in the *COMMANDS* section.

Causes the shell to notify the user asynchronously when the status of the

current (*job* not specified) or specified jobs changes; normally notification is presented before a prompt. This is automatic if the shell variable *notify* is set.

Controls the action of the shell on interrupts.

With no arguments, *onintr* restores the default action of the shell on interrupts, which action is to terminate shell scripts or return to the terminal command input level. If **-** is specified, all interrupts are ignored. If a *label* is given, the shell executes a *goto label* when an interrupt is received or a child process terminates because it was interrupted.

If the shell is running in the background and interrupts are being ignored, *onintr* has no effect; interrupts continue to be ignored by the shell and all invoked commands.

Pops the directory stack, returning to the new top directory.

With an argument, discards the *n* th entry in the stack. The elements of the directory stack are numbered from 0 starting at the top. A synonym for called is provided for historical reasons. Its use is not recommended because it is not part of the standard BSD **csh** and may not be supported in future releases.

With no arguments,

pushd exchanges the top two elements of the directory stack. Given a *name* argument, *pushd* changes to the new directory (using *cd*) and pushes the old current working directory (as in *csw*) onto the directory stack. With a numeric argument, *pushd* rotates the *n* th argument of the directory stack around to be the top element and changes to that directory. The members of the directory stack are numbered from the top starting at 0.

A synonym for *pushd,* called *gd,* is provided for historical reasons. Its use is not recommended since it is not part of the standard BSD **csh** and may not be supported in future releases.

rehash Causes the internal hash table of the contents of the directories in the *path* variable to be recomputed. This is needed if new commands are added to directories in the *path* while you are logged in. This should only be necessary if you add commands to one of your own directories or if a systems programmer changes the contents of one of the system directories.

repeat count command

The specified *command* (which is subject to the same restrictions as the *command* in the one-line **if** statement above) is executed *count* times. I/O redirections occur exactly once, even if *count* is 0.

set
set name
set name = word
set name [*index*]= *word*
set name =(wordlist)

The first form of **set** shows the value of all shell variables. Variables whose value is other than a single word print as a parenthesized word list. The second form sets *name* to the null string. The third form sets *name* to the single *word*. The fourth form sets the *index*th component of *name* to *word*; this component must already exist. The final form sets *name* to the list of words in *wordlist*. In all cases the value is command and filename expanded.

These arguments can be repeated to set multiple values in a single *set* command. Note, however, that variable expansion happens for all arguments before any setting occurs.

setenv name value

Sets the value of environment variable *name* to be *value*, a single string. The most commonly used environment variables, and are automatically imported to and exported from the **csh** variables *user*, *term*, and *path*; there is no need to use **setenv** for these.

If no argument is given, the members of

argv are shifted to the left, discarding An error occurs if **argv** is not set or has less than two strings assigned to it. When *variable* is specified, *shift* performs the same function on the specified *variable*.

csh reads commands from *name*. **source** commands can be nested, but if nested too deeply the shell may run out of file descriptors. An error in a **source** at any level terminates all nested **source** commands. Normally, input during **source** commands is not placed on the history list. The **-h** option can be used to place commands in the history list without being executing them.

Stops the current (no argument) or specified jobs executing in the background.

suspend Causes **csh** to stop as if it had been sent a *suspend* signal. Since **csh** normally ignores *suspend* signals, this is the only way to suspend the shell. This command gives an error message if attempted from a login shell.

switch (string)
case str1 :
...
breaksw
...
default:
...

breaksw

endsw Each **case** label (*str1*) is successively matched against the specified *string* which is first command and file name expanded. The form of the **case** labels is the Pattern Matching Notation with the exception that non-matching lists in bracket expressions are not supported (see *regexp*(5)). If none of the labels match before a **default** label is found, the execution begins after the **default** label. Each **case** label and the **default** label must appear at the beginning of a line. The **breaksw** command causes execution to continue after the *endsw*. Otherwise, control may fall through **case** labels and **default** labels as in C. If no label matches and there is no default, execution continues after the

When *command* is not specified, a summary of time used by this shell and its children is printed. If specified, the simple *command* is timed and a time summary as described under the **time** variable is printed. If necessary, an extra shell is created to print the time statistic when the command completes.

The current file creation mask is displayed
(*value* not specified) or set to the specified *value*. The mask is given in octal. Common values for the mask are which gives all permissions to the owner and group and read and execute permissions to all others, or which gives all permissions to the owner, and only read and execute permission to the group and all others.

unalias pattern
All aliases whose names match the specified *pattern* are discarded. Thus, all aliases are removed by No error occurs if *pattern* does not match an existing alias.

unhash Use of the internal hash table to speed location of executed programs is disabled.

unset pattern
All variables whose names match the specified *pattern* are removed. Thus, all variables are removed by this has noticeably undesirable side-effects. No error occurs if *pattern* matches nothing.

unsetenv pattern
Removes all variables whose names match the specified *pattern* from the environment. See also the **setenv** command above and *printenv*(1).

wait Waits for all background jobs to terminate. If the shell is interactive, an interrupt can disrupt the wait, at which time the shell prints names and job numbers of all jobs known to be outstanding.

while (expression)
. . .
end While the specified *expression* evaluates non-zero, the commands between the **while** and the matching **end** are evaluated. **break** and **continue** can be used to terminate or continue the loop prematurely. (The **while** and **end** must appear alone on their input lines.) If the input is a terminal (i.e., not a script), prompting occurs the first time through the loop as for the **foreach** statement.

% job Brings the specified job into the foreground.

% job &
Continues the specified job in the background.

The first form prints the values of all the shell variables.
The second form sets the specified *name* to the value of *expression*. If the expression contains or at least this part of the expression must be placed within parentheses. The third form assigns the value of *expression* to the *index*th argument of *name*. Both *name* and its *index*th component must already exist.

The operators etc., are available as in C. White space can optionally separate the *name* from the assignment operator. However, spaces are mandatory in separating components of *expression* which would otherwise be single words.

Special postfix **++** and **- -** operators increment and decrement *name*, respectively (e.g.,

Non-Built-In Command Execution

When a command to be executed is not a built-in command, **csh** attempts to execute the command via *exec*(2). Each word in the variable *path* names a directory in which the shell attempts to find the command (if the command does not begin with If neither **-c** nor **-t** is given, the shell hashes the names in these directories into an internal table so that an *exec* is attempted only in those directories where the command might possibly reside. This greatly speeds command location when a large number of directories are present in the search path. If this mechanism has been turned off (via or if **-c** or **-t** was given, or if any directory component of *path* does not begin with a the shell concatenates the directory name and the given command name to form a path name of a file which it then attempts to execute.

Commands placed inside parentheses are always executed in a subshell. Thus

(cd ; pwd)

prints the *home* directory then returns to the current directory upon completion, whereas:

cd ; pwd

remains in the *home* directory upon completion.

When commands are placed inside parentheses, it is usually to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary file, it is assumed to be a script file, which is a file of data for an interpreter that is executed as a separate process.

csh first attempts to load and execute the script file (see *exec*(2)). If the first two characters of the script file are *exec*(2) expects an interpreter path name to follow and attempts to execute the specified interpreter as a separate process to read the entire script file.

If no *#! interpreter* is named, and there is an *alias* for the shell, the words of the *alias* are inserted at the beginning of the argument list to form the shell command. The first word of the *alias* should be the full path name of the command to be used. Note that this is a special, late-occurring case of *alias* substitution, which inserts words into the argument list without modification.

If no *#! interpreter* is named and there is no shell *alias*, but the first character of the file is the interpreter named by the **$shell** variable is executed (note that this normally would be unless the user has reset If **$shell** is not set, **/bin/csh** is executed.

If no *!# interpreter* is named, and there is no shell alias, and the first character of the file is not **/bin/sh** is executed to interpret the script file.

History Substitutions

History substitutions enable you to repeat commands, use words from previous commands as portions of new commands, repeat arguments of a previous command in the current command, and fix spelling or typing mistakes in an earlier command.

History substitutions begin with an exclamation point (**!**). Substitutions can begin anywhere in the input stream, but *cannot* be nested. The exclamation point can be preceded by a backslash to cancel its special meaning. For convenience, an exclamation point is passed to the parser unchanged when it is followed by a blank, tab, newline, equal sign, or left parenthesis. Any input line that contains history substitution is echoed on the terminal before it is executed for verification.

Commands input from the terminal that consist of one or more words are saved on the history list. The history substitutions reintroduce sequences of words from these saved commands into the input stream. The number of previous commands saved is controlled by the **history** variable. The previous command is always saved, regardless of its value. Commands are numbered sequentially from 1.

You can refer to previous events by event number (such as **!10** for event 10), relative event location (such as

!-2 for the second previous event), full or partial command name (such as **!d** for the last event using a command with initial character and string expression (such as **!?mic?** referring to an event containing the characters

These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case, **!!** is a re-do; it refers to the previous command.

To select words from a command, use a colon **(:)** and a designator for the desired words after the event specification. The words of an input line are numbered from zero. The basic word designators are:

0 First word (i.e., the command name itself).

n *n*th word.

ˆ First argument. (This is equivalent to

$ Last word.

a-*b* Range of words from *a* through *b*. Special cases are an abbreviation for ‘‘word 0 through word *y*’’; and *x* **- ,** which means ‘‘word *x* up to, but not including, word

***** Range from the second word through the last word.

% Used with a search sequence to substitute the immediately preceding matching word.

The colon separating the command specification from the word designator can be omitted if the argument selector begins with a or

After word designator can be followed by a sequence of modifiers, each preceded by a colon. The following modifiers are defined:

h Use only the first component of a path name by removing all following components.

r Use the root file name by removing any trailing suffix (.xxx).

e Use the file name’s trailing suffix **(** *.xxx* **)** by removing the root name.

s / l / r
substitute the value of *r* for the value *l* in the indicated command.

t Use only the final file name of a path name by removing all leading path name components.

& Repeat the previous substitution.

p Print the new command but do not execute it.

q Quote the substituted words, preventing further substitutions.

x Like but break into words at blanks, tabs and newlines.

g Use a command as a prefix to another modifier to cause the specified change to be made globally. All words in the command are changed, one change per word, and each string enclosed in single quotes **(’)** or double quotes is treated as a single word.

Unless preceded by a the modification is applied only to the first modifiable word. An error results if a substitution is attempted and cannot be completed (i.e., if you ask for a substitution of **!11** on a history buffer containing only 10 commands).

The left hand side of substitutions are strings; not regular expressions in the sense of HP-UX editors. Any character can be used as the delimiter in place of a slash (**/**). Use a backslash to quote a delimiter character if it is used in the *l* or *r* string. The character **&** in the right-hand side is replaced by the text from the left. A **** also quotes A null *l* string uses the previous string either from an *l* or from a contextual scan string *s* in *!? s ?*. The trailing delimiter in the substitution can be omitted if a new-line character follows immediately, as may the trailing **?** in a contextual scan.

A history reference can be given without an event specification (as in In this case, the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus

!?foo?^ !$

gives the first and last arguments from the command matching

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a circumflex (**^**). This is equivalent to providing a convenient shorthand for substitutions on the text of the previous line. Thus **^lb^lib** fixes the spelling of **lib** in the previous command.

Finally, a history substitution can be enclosed within curly braces **{ }** if necessary to insulate it from the characters which follow. Thus, after

ls -ld ~paul

one could execute **!{l}a** to do

ls -ld ~paula

while **!la** would look for a command starting with

Quoting with Single and Double Quotes

The quotation of strings by single quotes (**'**) and double quotes can be used to prevent all or some of the remaining substitutions. Strings enclosed in single quotes are protected from any further interpretation. Strings enclosed in double quotes are still variable- and command-expanded as described below.

In both cases the resulting text becomes (all or part of) a single word. Only in one special case (see *Command Substitution* below) does a double-quoted string yield parts of more than one word; single-quoted strings never do.

Alias Substitution

csh maintains a list of aliases that can be established, displayed, and modified by the **alias** and **unalias** commands. After a command line is scanned, it is parsed into distinct commands and the first word of each command, left-to-right, is checked to see if it has an alias. If it does, the text which is the alias for that command is reread with the history mechanism available as if that command was the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, the argument list is left unchanged.

Thus, if the alias for **ls** is the command **ls /usr** maps to leaving the argument list undisturbed. Similarly, if the alias for **lookup** was **lookup bill** maps to **grep bill /etc/passwd .**

If an alias is found, the word transformation of the input text is performed and the aliasing process begins again on the re-formed input line. Looping is prevented if the first word of the new text is the same as the old by flagging it to prevent further aliasing. Other loops are detected and cause an error.

Note that the mechanism allows aliases to introduce parser metasyntax. Thus:

alias print 'pr \!* | lp'

makes a command that uses *pr*(1) to print its arguments on the line printer.

Expressions

Some of the built-in commands take expressions in which the operators are similar to those of C, with the same precedence. These expressions appear in the and **while** commands. The following operators are available (shown in order of increasing precedence):

|| && | ^ & == != =~ !~ <= >= < > << >> + - * / % ! ~ ( )

The following list shows the grouping of these operators. The precedence decreases from top to bottom in the list:

*** / %**
+ -
<< >>
<= >= < >
== != =~ !~

The operators and compare their arguments as strings; all others operate on numbers. The operators **=~** and

!~ are similar to **!=** and except that the right-hand side is a *pattern* (containing and instances of against which the left hand operand is matched. This reduces the need for use of the **switch** statement in shell scripts when all that is really needed is pattern matching.

Strings beginning with **0** are considered octal numbers. Null or missing arguments are considered The result of all expressions are strings that represent decimal numbers. It is important to note that no two components of an expression can appear in the same word. These components should be surrounded by spaces except when adjacent to components of expressions that are syntactically significant to the parser: and

Also available in expressions as primitive operands are command executions enclosed in curly braces (**{ }**) and file enquiries of the form where *l* is one of:

- **r** read access
- **w** write access
- **x** execute access
- **e** existence
- **o** ownership
- **z** zero size
- **f** plain file
- **d** directory

The specified *filename* is command- and file-name expanded then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible, all inquiries return false (0). Command executions succeed, returning true, if the command exits with status 0; otherwise they fail, returning false. If more detailed status information is required, the command should be executed outside of an expression and the **status** variable examined.

Control of the Flow

csh contains a number of commands that can be used to regulate the flow of control in command files (shell scripts) and (in limited but useful ways) from terminal input. These commands all operate by forcing the shell to reread or skip parts of its input and, due to the implementation, restrict the placement of some of the commands.

The and **while** statements, as well as the form of the **if** statement require that the major keywords appear in a single simple command on an input line as shown below.

If the shell's input is not seekable, the shell buffers input whenever a loop is being read and performs seeks in this internal buffer to accomplish the rereading implied by the loop. (To the extent that this allows, backward succeed on non-seekable inputs.)

Signal Handling

csh normally ignores *quit* signals. Jobs running in background mode are immune to signals generated from the keyboard, including hangups. Other signals have the values which the shell inherited from its parent. handling of interrupts and terminate signals in shell scripts can be controlled by *onintr*. Login shells catch the *terminate* signal; otherwise this signal is passed on to children from the state in the shell's parent. In no case are interrupts allowed when a login shell is reading the file

Command Line Parsing

csh splits input lines into words at blanks and tabs. The following exceptions (parser metacharacters) are considered separate words:

- **&** ampersand;
- **|** vertical bar;
- **;** semicolon;
- **<** less-than sign;
- **>** greater-than sign;
- **(** left parenthesis;
- **)** right parenthesis;
- **&&** double ampersand;

||	double vertical bar;
<<	double less-than sign;
>>	double greater-than sign;
#	comment delimiter

The backslash **(\ )** removes the special meaning of these parser metacharacters. A parser metacharacter preceded by a backslash is interpreted as its ASCII value. A newline character (ASCII 10) preceded by a backslash is equivalent to a blank.

Strings enclosed in single or double quotes form parts of a word. Metacharacters in these strings, including blanks and tabs, do not form separate words. Within pairs of backslashes or quotes, a newline preceded by a backslash gives a true newline character.

When input is not a terminal, the **#** charactercharacter introduces a comment terminated by a newline.

CSH VARIABLES

csh maintains a set of variables. Each variable has a value equal to zero or more strings (words). Variables have names consisting of up to 20 letters and digits starting with a letter. The underscore character is considered a letter. The value of a variable may be displayed and changed by using the **set** and **unset** commands. Some of the variables are Boolean, that is, the shell does not care what their value is, only whether they are set or not.

Some operations treat variables numerically. The at sign command permits numeric calculations to be performed and the result assigned to a variable. The null string is considered to be zero, and any subsequent words of multi-word values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable expansion is performed keyed by the dollar sign **($) character.** Variable expansion can be prevented by preceding the dollar sign with a backslash character **(\ )** except within double quotes where substitution **always** occurs. Variables are never expanded if enclosed in single quotes. Strings quoted by single quotes are interpreted later (see *Command Substitution*) so variable substitution does not occur there until later, if at all. A dollar sign is passed unchanged if followed by a blank, tab, or end-of-line.

Input/output redirections are recognized before variable expansion, and are variable expanded separately. Otherwise, the command name and entire argument list are expanded together.

Unless enclosed in double quotes or given the **:q** modifier, the results of variable substitution may eventually be command and file name substituted. Within double quotes, a variable whose value consists of multiple words expands to a portion of a single word, with the words of the variable's value separated by blanks. When the **:q** modifier is applied to a substitution, the variable expands to multiple words with each word separated by a blank and quoted to prevent later command or file name substitution.

The following metasequences are provided for introducing variable values into the shell input. Except as noted, it is an error to reference a variable that is not set.

$ variable_name
${ variable_name }

When interpreted, this sequence is replaced by the words of the value of the variable *variable_name*, each separated by a blank. Braces insulate *variable_name* from subsequent characters that would otherwise be interpreted to be part of the variable name itself.
If *variable_name* is not a **csh** variable, but is set in the environment, that value is used. **Non- csh** variables cannot be modified as shown below.

$ variable_name[*selector*]
${ variable_name[*selector*] *}*

This modification selects only some of the words from the value of *variable_name*. The selector is subjected to variable substitution, and can consist of a single number or two numbers separated by a dash. The first word of a variable's value is numbered If the first number of a range is omitted it defaults to If the last member of a range is omitted it defaults to the total number of words in the variable **($#** *variable_name***).** An asterisk metacharacter used as a selector selects all words.

$# variable_name
${# variable_name }
This form gives the number of words in the variable, and is useful for forms using a [*selector*] option.

$0 This form substitutes the name of the file from which command input is being read. An error occurs if the file name is not known.

$ number
${ number }
This form is equivalent to an indexed selection from the variable **argv ($argv[** *number* **]).**

$* This is equivalent to selecting all of *argv* **($argv[*]).**

The modifiers and can be applied to the substitutions above, as can CR :gh , CR :gt , and CR :gr . If curly braces **({ })** appear in the command form, the modifiers must appear within the braces. *The current implementation allows only one* **:** *modifier on each* **$d** *expansion*.

The following substitutions cannot be modified with **:** modifiers:

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

$?0 Substitutes **1** if the current input file name is known, **0** if it is not.

$$ Substitutes the (decimal) 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 shell script.

Pre-Defined and Environment Variables

The following variables have special meaning to the shell. Of these and **status** are always set by the shell. Except for **cwd** and this setting occurs only at initialization (initial execution of These variables are not modified unless modified explicitly by the user.

csh copies the HP-UX environment variable USER into the shell variable the environment variable **TERM** into the environment variable **HOME** into and **PATH** into **csh** copies these values back into the environment whenever the **csh** variables are reset.

In a windowed environment, if **csh** detects that the window has changed size, **csh** sets the environment variables **LINES** and **COLUMNS** to match the new window size.

argv This variable is set to the arguments of the **csh** command statement. It is from this variable that positional parameters are substituted; i.e., **$1** is replaced by etc.

cdpath This variable gives a list of alternate directories searched to find subdirectories in *chdir* commands.

cwd This variable contains the absolute path name of the current working directory. Whenever changing directories (using *cd*), this variable is updated.

echo This variable is set by the **-x** command line option. If set, all built-in commands and their arguments are echoed to the standard output device just before being executed. Built-in commands are echoed before command and file name substitution, since these substitutions are then done selectively. For non-built-in commands, all expansions occur before echoing.

hidden If set, **csh** includes unmatched CDFs in expansions (see *cdf*(4)). Also, unmatched CDFs are displayed with a trailing **+** when using the **control-D** command to display matching file names (see the *CSH UTILITIES* section). By default, **hidden** is not set.

history This variable is used to create the command history buffer and to set its size. If this variable is not set, no command history is maintained and history substitutions cannot be made. Very large values of **history** can cause shell memory overflow. Values of 10 or 20 are normal. All commands, executable or not, are saved in the command history buffer.

home This variable contains the absolute path name to your home directory. The variable **home** is initialized from the HP-UX environment. File name expansion of tilde (˜) refers to this variable.

ignoreeof If set, **csh** ignores end-of-file characters from input devices that are terminals. **csh** exits normally when it encounters the end-of-file condition **(CTRL- D** typed as the first character on a command line). Setting *ignoreeof* prevents the current shell from being killed by an accidental **(CTRL- D .** However, to prevent an infinite loop of EOF input, **csh** terminates if it receives 26 consecutive EOFs.

mail This variable contains a list of the files where **csh** checks for your mail. **csh** periodically (default is 10 minutes) checks this variable before producing a prompt upon command completion. If the variable contains a file name that has been modified since the last check (resulting from mail being put in the file), **csh** prints

If the first word of the value of **mail** is numeric, that number specifies a different mail checking interval in seconds.

If multiple mail files are specified, the shell says **New mail in** *file_name*, where *file_name* is the file containing the mail.

noclobber This variable places restrictions on output redirection to ensure that files are not accidentally destroyed, and that commands using append redirection **(>>)** refer to existing files.

noglob If set, file name expansion is inhibited. This is most useful in shell scripts that are not dealing with file names, or after a list of file names has been obtained and further expansions are not desirable.

nonomatch If set, it is no longer an error for a file name expansion to not match any existing files. If there is no match, the primitive pattern is returned. It is still an error for the primitive pattern to be malformed. For example, **'echo ['** still gives an error.

notify If set, **csh** notifies you immediately (through your standard output device) of background job completions. The default is **unset** (indicate job completions just before printing a prompt).

path Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies your current working directory. If there is no *path* variable, only full path names can be executed. When *path* is not set and when users do not specify full path names, **csh** searches for the command through the directories **.** (current directory), and A **csh** which is given neither the **-c** nor the **-t** option normally hashes the contents of the directories in the **path** variable after reading and each time the **path** variable is reset. If new commands are added to these directories while the shell is active, it is necessary to execute **rehash** for **csh** to access these new commands.

prompt This variable lets you select your own prompt character string. The prompt is printed before each command is read from an interactive terminal input. If a **!** appears in the string, it is replaced by the current command history buffer event number unless a preceding **** is given. The default prompt is the percent sign (**%**) for users and the **#** character for the super-user.

shell This variable contains the name of the file in which the **csh** program resides. This variable is used in forking shells to interpret files that have their execute bits set but which are not executable by the system. (See the description of *Non-Built-In Command Execution*).

status This variable contains the status value returned by the last command. If the command terminated abnormally, 0200 is added to the status variable's value. Built-in commands which terminated abnormally return exit status and all other built-in commands set status to

time This variable contains a numeric value that controls the automatic timing of commands. If set, **csh** prints, for any command taking more than the specified number of cpu seconds, a line of information to the standard output device giving user, system, and real execution times plus a utilization percentage. The utilization percentage is the ratio of user plus system times to real time. This message is printed after the command finishes execution.

verbose This variable is set by the **-v** command line option. If set, the words of each command are printed on the standard output device after history substitutions have been made.

Command and File name Substitution

The remaining substitutions, command and file name substitution, are applied selectively to the arguments of built-in commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands which are not internal to the shell, the command name is substituted separately from the argument list. This occurs very late, after input-output redirection is performed, and in a child of the main shell.

Command Substitution

Command substitution is indicated by a command enclosed in grave accents (` . . . `). The output from such a command is normally broken into separate words at blanks, tabs and newlines, with null words being discarded; this text then replacing the original string. Within double quotes, only newlines force new words; blanks and tabs are preserved.

In any case, the single final newline does not force a new word. Note that it is thus possible for a command substitution to yield only part of a word, even if the command outputs a complete line.

File name Substitution

Each command *word* is processed as a pattern for file name substitution, also known as **globbing**, and replaced with a sorted list of file names which match the pattern. The form of the patterns is the Pattern Matching Notation defined by *regexp*(5) with the following exceptions:

- Non-matching lists in bracket expressions are not supported.
- In a list of words specifying file name substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match.
- The metanotation **a{b,c,d}e** is a shorthand for "abe ace ade". Left to right order is preserved, with results of matches being sorted separately at a low level to preserve this order. This construct may be nested. Thus:

  ~source/s1/{oldls,ls}.c

  expands to

  /usr/source/s1/oldls.c /usr/source/s1/ls.c

  whether or not these files exist, without any chance of error if the home directory for **source** is Similarly,

  ../{memo,*box}

  might expand to

../memo ../box ../mbox

(Note that **memo** was not sorted with the results of matching As a special case, and **{ }** are passed undisturbed.

Input/Output

The standard input and standard output of a command can be redirected with the following syntax:

< name Open file *name* (which is first variable, command and file name expanded) as the standard input.

<< word Read the shell input up to a line which is identical to *word*. *word* is not subjected to variable, file name or command substitution, and each input line is compared to *word* before any substitutions are done on this input line. Unless a quoting or ` appears in *word*, variable and command substitution is performed on the intervening lines, allowing **\\** to quote **\\** and `. Commands which are substituted have all blanks, tabs, and newlines preserved, except for the final newline which is dropped. The resultant text is placed in an anonymous temporary file which is given to the command as standard input.

> name
>! name
>& name
>&! name The file *name* is used as standard output. If the file does not exist, it is created; if the file exists, it is truncated, and its previous contents are lost.

If the variable **noclobber** is set, the file must not exist or be a character special file (e.g., a terminal or or an error results. This helps prevent accidental destruction of files. In this case the exclamation point (**!**) forms can be used to suppress this check.

The forms involving the ampersand character (**&**) route the standard error into the specified file as well as the standard output. *name* is expanded in the same way as < input file names are.

>> name
>>& name
>>! name
>>&! name Uses file *name* as standard output the same as but appends output to the end of the file. If the variable **noclobber** is set, it is an error for the file not to exist unless one of the **!** forms is given. Otherwise, it is similar to

A command receives the environment in which the shell was invoked as modified by the input-output parameters and the presence of the command in a pipeline. Thus, unlike some previous shells, commands executed from a shell script have no access to the text of the commands by default; rather they receive the original standard input of the shell. The **<<** mechanism should be used to present inline data. This permits shell scripts to function as components of pipelines and allows the shell to block-read its input.

Diagnostic output can be directed through a pipe with the standard output. Simply use the form **|&** rather than **|** by itself.

CSH UTILITIES

File Name Completion

In typing file names as arguments to commands, it is no longer necessary to type a complete name, only a unique abbreviation is necessary. When you want the system to try to match your abbreviation, press the ESC key. The system then completes the file name for you, echoing the full name on your terminal. If the abbreviation does not match an available file name, the terminal's bell is sounded. The file name may be partially completed if the prefix matches several longer file names. In this case, the name is extended up to the ambiguous deviation, and the bell is sounded.

File name completion works equally well when other directories are addressed. In addition, the tilde (˜) convention for home directories is understood in this context.

Viewing a File or Directory List

At any point in typing a command, you can request "what files are available" or "what files match my current specification". Thus, when you have typed:

% cd ˜speech/data/bench/fritz/

you may wish to know what files or subdirectories exist (in without aborting the command you are typing. Typing **CTRL- D** at this point lists the files available. Files are listed in multicolumn format, sorted by column. Directories and executable files are identified by a trailing **/** and respectively. Once printed, the command is re-echoed for you to complete. Additionally, you may want to know which files match a prefix, the current file specification so far. If you had typed:

% cd ˜speech/data/bench/fr

followed by a **CTRL- D ,** all files and subdirectories whose prefix was **fr** in the directory **˜speech/data/bench** would be printed. Notice that the example before was simply a degenerate case of this with a null trailing file name. (The null string is a prefix of all strings.) Notice also that a trailing slash is required to pass to a new sub-directory for both file name completion and listing. Note that the degenerate case

% ˜ˆD

prints a full list of login names on the current system.

Command Name Recognition

Command name recognition and completion works in the same manner as file name recognition and completion above. The current value of the environment variable **PATH** is used in searching for the command. For example

% newa [Escape]

might expand to

% newaliases

Also,

% new [Control]-[D]

lists all commands (along that begin with As an option, if the shell variable **listpathnum** is set, a number indicating the index in **PATH** is printed next to each command on a [Control]-[D] listing.

Autologout

A new shell variable has been added called If the terminal remains idle (no character input) at the shell's top level for a number of minutes greater than the value assigned to you are automatically logged off. The **autologout** feature is temporarily disabled while a command is executing. The initial value of **autologout** is 60. If unset or set to 0, **autologout** is entirely disabled.

Command Line Control

A **ˆR** re-prints the current command line; **ˆW** erases the last word entered on the current command line.

Sanity

C shell restores your terminal to a sane mode if it appears to return from some command in raw, cbreak, or noecho mode.

Saving Your History Buffer

csh has the ability to save your history list between login sessions. If the shell variable **savehist** is set to a number, that number of command events from your history list is saved. For example, placing the line

set history=10 savehist=10

in your **.cshrc** file maintains a history buffer of length 10 and saves the entire list when you logout. When you log back in, the entire buffer is restored. The commands are saved in the file **.history** in your login directory.

EXTERNAL INFLUENCES

Environment Variables

LC_COLLATE determines the collating sequence used in evaluating pattern matching notation for file name substitution.

LC_CTYPE determines the interpretation of text as single and/or multi-byte characters, the classification of characters as letters, and the characters matched by character class expressions in pattern matching notation.

LANG determines the language in which messages are displayed.

If LC_COLLATE or LC_CTYPE is not specified in the environment or is set to the empty string, the value of LANG is used as a default for each unspecified or empty variable. If LANG is not specified or is set to the empty string, a default of "C" (see *lang*(5)) is used instead of LANG. If any internationalization variable contains an invalid setting, **csh** behaves as if all internationalization variables are set to "C". See *environ*(5).

International Code Set Support

Single- and multi-byte character code sets are supported.

WARNINGS

The **.cshrc** file should be structured such that it cannot generate any output on standard output or standard error, including occasions when it is invoked without an affiliated terminal. *rcp*(1) causes **.cshrc** to be sourced, and any output generated by this file, even to standard error causes problems. Commands such as *stty*(1) should be placed in not in so that their output cannot affect *rcp*(1).

csh has certain limitations. Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command which involves file name expansion is limited to one-sixth the number of characters allowed in an argument list. Command substitutions may substitute no more characters than are allowed in an 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, **csh** prints the directory it started in if it is different from the current directory; this can be misleading (i.e., wrong) because 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 interrupt b, the shell then immediately executes c. This is especially noticeable if this expansion results from an It suffices to place the sequence of commands in parentheses to force it into a subshell; i.e.,

Because of the signal handling required by csh, interrupts are disabled just before a command is executed, and restored as the command begins execution. There may be a few seconds delay between when a command is given and when interrupts are recognized.

Control over tty output after processes are started is primitive; perhaps this will inspire someone to work on a good virtual terminal interface. In a virtual terminal interface much more interesting things could be done with output control.

Alias substitution is most often used to clumsily simulate shell procedures; shell procedures should be provided rather than aliases.

Commands within loops, prompted for by are not placed in the *history* list. Control structure 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. All and more than one **:** modifier should be allowed on **$** substitutions.

Terminal type is examined only the first time you attempt recognition.

To list all commands on the system along enter [Space]-[Ctrl]-[D].

The csh metasequence **!~** does not work.

In an international environment, character ordering is determined by the setting of LC_COLLATE, rather than by the binary ordering of character values in the machine collating sequence. This brings with it certain attendant dangers, particulary when using range expressions in file name generation patterns. For example, the command,

rm [a-z]*

might be expected to match all file names beginning with a lowercase alphabetic character. However, if dictionary ordering is specified by LC_COLLATE, it would also match file names beginning with an uppercase character (as well as those beginning with accented letters). Conversely, it would fail to match letters collated after **z** in languages such as Norwegian.

The correct (and safe) way to match specific character classes in an international environment is to use a pattern of the form:

rm [[:lower:]]*

This uses **LC_CTYPE** to determine character classes and works predictably for all supported languages and codesets. For shell scripts produced on non-internationalized systems (or without consideration for the above dangers), it is recommended that they be executed in a non-NLS environment. This requires that etc., be set to "C" or not set at all.

csh implements command substitution by creating a pipe between itself and the command. If the root file system is full, the substituted command cannot write to the pipe. As a result, the shell receives no input from the command, and the result of the substitution is null. In particular, using command substitution for variable assignment under such circumstances results in the variable being silently assigned a NULL value.

Relative path changes (such as when in a symbolically linked directory, cause knowledge of the working directory to be along the symbolic path instead of the physical path.

Prior to HP-UX Release 9.0, when getting its input from a file, would exit immediately if unable to execute a command (such as if it was unable to find the command). Beginning at Release 9.0, **csh** continues on and attempts to execute the remaining commands in the file. However, if the old behavior is desired for compatibility purposes, set the environment variable **EXITONERR** to 1.

AUTHOR

csh was developed by the University of California, Berkeley and HP.

FILES

~/.cshrc	A csh script sourced (executed) at the beginning of execution by each shell. See WARNINGS
~/.login	A csh script sourced (executed) by login shell, after **.cshrc** at login.
~/.logout	A csh script sourced (executed) by login shell, at logout.
/etc/passwd	Source of home directories for
/bin/sh	Standard shell, for shell scripts not starting with a
/etc/csh.login	A csh script sourced (executed) before **~/.cshrc** and **~/.login** when starting a csh login (analogous to **/etc/profile** in the Bourne shell).
/tmp/sh*	Temporary file for

SEE ALSO

sh(1), access(2), exec(2), fork(2), pipe(2), umask(2), wait(2), tty(7), a.out(4), environ(5), lang(5), regexp(5).

C Shell tutorial in *Shells Users Guide* .