NAME

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

SYNOPSIS

csh [-cefinstvVxX] [arg ...]

DESCRIPTION

Csh is a command language interpreter. It begins by executing commands from the file `.cshrc' in the home directory of the invoker. If this is a login shell then it also executes commands from the file `.login' there. In the normal case, the shell will then begin reading commands from the terminal, prompting with `% '. Processing of arguments and the use of the shell to process files containing command scripts will be described later.

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 command history list and then parsed. Finally each command in the current line is executed.

When a login shell terminates it executes commands from the file `.logout' in the users home directory.

Lexical structure

The shell splits input lines into words at blanks and tabs with the following exceptions. The characters `&' `|' `;' `<' `>' `(' `)' form separate words. If doubled in `&&', `||', `<<' or `>>' these pairs form single words. These parser metacharacters may be made part of other words, or prevented their special meaning, by preceding them with `\'. A newline preceded by a `\' is equivalent to a blank.

In addition strings enclosed in matched pairs of quotations, `'', `\' or `"', form parts of a word; metacharacters in these strings, including blanks and tabs, do not form separate words. These quotations have semantics to be described subsequently. Within pairs of `' or `"' characters a newline preceded by a `\' gives a true newline character.

When the shell's input is not a terminal, the character `#' introduces a comment which continues to the end of the input line. It is prevented this special meaning when preceded by `\' and in quotations using `\', `'', and `"'.

Commands

A simple command is a sequence of words, the first of which specifies the command to be executed. A simple command or a sequence of simple commands separated by `|' characters forms a pipeline. The output of each command in a pipeline is connected to the input of the next. Sequences of pipelines may be

separated by `;', and are then executed sequentially. A sequence of pipelines may be executed without waiting for it to terminate by following it with an `&'. Such a sequence is automatically prevented from being terminated by a hangup signal; the *nohup* command need not be used.

Any of the above may be placed in `(' `)' to form a simple command (which may be a component of a pipeline, etc.) It is also possible to separate pipelines with `||' or `&&' indicating, as in the C language, that the second is to be executed only if the first fails or succeeds respectively. (See *Expressions*.)

Substitutions

We now describe the various transformations the shell performs on the input in the order in which they occur.

History substitutions

History substitutions can be used to reintroduce sequences of words from previous commands, possibly performing modifications on these words. Thus history substitutions provide a generalization of a *redo* function.

History substitutions begin with the character `!' and may begin **anywhere** in the input stream if a history substitution is not already in progress. This `!' may be preceded by an `\' to prevent its special meaning; a `!' is passed unchanged when it is followed by a blank, tab, newline, `=' or `('. History substitutions also occur when an input line begins with `^'. This special abbreviation will be described later.

Any input line which contains history substitution is echoed on the terminal before it is executed as it could have been typed without history substitution.

Commands input from the terminal which consist of one or more words are saved on the history list, the size of which is controlled by the *history* variable. The previous command is always retained. Commands are numbered sequentially from 1.

For definiteness, consider the following output from the history command:

```
 9  write michael
10  ex write.c
11  cat oldwrite.c
12  diff *write.c
```

The commands are shown with their event numbers. It is not usually necessary to use event numbers, but the current event number can be made part of the *prompt* by placing an `!' in the prompt

string.

With the current event 13 we can refer to previous events by event number `!11', relatively as in `!-2' (referring to the same event), by a prefix of a command word as in `!d' for event 12 or `!w' for event 9, or by a string contained in a word in the command as in `!?mic?' also referring to event 9. These forms, without further modification, simply reintroduce the words of the specified events, each separated by a single blank. As a special case `!!' refers to the previous command; thus `!!' alone is essentially a _redo_. The form `!#' references the current command (the one being typed in). It allows a word to be selected from further left in the line, to avoid retyping a long name, as in `!#:1'.

To select words from an event we can follow the event specification by a `:' and a designator for the desired words. The words of a input line are numbered from 0, the first (usually command) word being 0, the second word (first argument) being 1, etc. The basic word designators are:

0	first (command) word
_n_	_n_'th argument
^	first argument, i.e. `1'
$	last argument
%	word matched by (immediately preceding) ?_s_? search
_x-y_	range of words
-_y_	abbreviates `0-_y_'
*	abbreviates `^-$', or nothing if only 1 word in event
_x_*	abbreviates `_x_-$'
_x_-	like `_x_*' but omitting word `$'

The `:' separating the event specification from the word designator can be omitted if the argument selector begins with a `^', `$', `*' `-' or `%'. After the optional word designator can be placed a sequence of modifiers, each preceded by a `:'. The following modifiers are defined:

h	Remove a trailing pathname component, leaving the head.
r	Remove a trailing `.xxx' component, leaving the root name.
s/_l_/_r_/	Substitute _l_ for _r_
t	Remove all leading pathname components, leaving the tail.
&	Repeat the previous substitution.
g	Apply the change globally, prefixing the above, e.g. `g&'.
p	Print the new command but do not execute it.
q	Quote the substituted words, preventing further substitutic
x	Like q, but break into words at blanks, tabs and newlines.

Unless preceded by a `g' the modification is applied only to the first modifiable word. In any case it is an error for no word to be applicable.

The left hand side of substitutions are not regular expressions in the sense of the editors, but rather strings. Any character may be used as the delimiter in place of `/'; a `\' quotes the delimiter into the *l* and *r* strings. The character `&' in the right hand side is replaced by the text from the left. A `\' quotes `&' also. A null *l* uses the previous string either from a *l* or from a contextual scan string *s* in `!?*s*?'. The trailing delimiter in the substitution may be omitted if a newline follows immediately as may the trailing `?' in a contextual scan.

A history reference may be given without an event specification, e.g. `!$'. In this case the reference is to the previous command unless a previous history reference occurred on the same line in which case this form repeats the previous reference. Thus `!?foo?↑ !$' gives the first and last arguments from the command matching `?foo?'.

A special abbreviation of a history reference occurs when the first non-blank character of an input line is a `↑'. This is equivalent to `!:s↑' 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 may be surrounded with `{' and `}' if necessary to insulate it from the characters which follow. Thus, after `ls -ld ~paul' we might do `!{l}a' to do `ls -ld ~paula', while `!la' would look for a command starting `la'.

Quotations with ' and "

The quotation of strings by `'' and `"' can be used to prevent all or some of the remaining substitutions. Strings enclosed in `'' are prevented any further interpretation. Strings enclosed in `"' are yet 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 Substitition* below) does a `"' quoted string yield parts of more than one word; `' quoted strings never do.

Alias substitution

The shell maintains a list of aliases which 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, then the text which is the alias for that command is reread with the history mechanism available as though that command were the previous input line. The resulting words replace the command and argument list. If no reference is made to the history list, then the argument list is left unchanged.

Thus if the alias for `ls' is `ls -l' the command `ls /usr' would map to `ls -l /usr', the argument list here being undisturbed. Similarly if the alias for `lookup' was `grep !↑ /etc/passwd' then `lookup bill' would map 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 reformed 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 we can `alias print 'pr \!* | lpr'' to make a command which *pr*'s its arguments to the line printer.

Variable substitution

The shell maintains a set of variables, each of which has as value a list of zero or more words. Some of these variables are set by the shell or referred to by it. For instance, the *argv* variable is an image of the shell's argument list, and words of this variable's value are referred to in special ways.

The values of variables may be displayed and changed by using the *set* and *unset* commands. Of the variables referred to by the shell a number are toggles; the shell does not care what their value is, only whether they are set or not. For instance, the *verbose* variable is a toggle which causes command input to be echoed. The setting of this variable results from the **-v** command line option.

Other operations treat variables numerically. The `@' command permits numeric calculations to be performed and the result assigned to a variable. Variable values are, however, always represented as (zero or more) strings. For the purposes of numeric operations, the null string is considered to be zero, and the second and subsequent words of multiword values are ignored.

After the input line is aliased and parsed, and before each command is executed, variable substitution is performed keyed by `$' characters. This expansion can be prevented by preceding the `$' with a `\' except within `"'s where it **always** occurs, and within `''s where it **never** occurs. Strings quoted by ``' are interpreted later (see *Command substitution* below) so `$' substitution does not occur there until later, if at all. A `$' 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. It is thus possible for the first (command) word to this point to generate more than one word, the first of which becomes the command

name, and the rest of which become arguments.

Unless enclosed in `"' or given the `:q' modifier the results of variable substitution may eventually be command and filename substituted. Within `"' a variable whose value consists of multiple words expands to a (portion of) a single word, with the words of the variables value separated by blanks. When the `:q' modifier is applied to a substitution the variable will expand to multiple words with each word separated by a blank and quoted to prevent later command or filename 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 which is not set.

$name
${name}

> Are replaced by the words of the value of variable *name*, each separated by a blank. Braces insulate *name* from following characters which would otherwise be part of it. Shell variables have names consisting of up to 20 letters, digits, and underscores.

If *name* is not a shell variable, but is set in the environment, then that value is returned (but : modifiers and the other forms given below are not available in this case).

$name[selector]
${name[selector]}

> May be used to select only some of the words from the value of *name*. The selector is subjected to `$' substitution and may consist of a single number or two numbers separated by a `-'. The first word of a variables value is numbered `1'. If the first number of a range is omitted it defaults to `1'. If the last member of a range is omitted it defaults to `$#name'. The selector `*' selects all words. It is not an error for a range to be empty if the second argument is omitted or in range.

$#name
${#name}

> Gives the number of words in the variable. This is useful for later use in a `[selector]'.

$0

> Substitutes the name of the file from which command input is being read. An error occurs if the name is not known.

$number
${number}

> Equivalent to `$argv[number]'.

$*

Equivalent to `$argv[*]'.

The modifiers `:h', `:t', `:r', `:q' and `:x' may be applied to the substitutions above as may `:gh', `:gt' and `:gr'. If braces `{' '}' appear in the command form then the modifiers must appear within the braces. **The current implementation allows only one `:' modifier on each `$' expansion.**

The following substitutions may not be modified with `:' modifiers.

$?name
${?name}

Substitutes the string `1' if name is set, `0' if it is not.

$?0

Substitutes `1' if the current input filename is know, `0' if it is not.

$$

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

Command and filename substitution

The remaining substitutions, command and filename substitution, are applied selectively to the arguments of builtin commands. This means that portions of expressions which 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 `\`'. 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 `"'s, 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.

Filename substitution

If a word contains any of the characters `*', `?', `[' or `{' or begins with the character `~', then that word is a candidate for

filename substitution, also known as `globbing'. This word is then regarded as a pattern, and replaced with an alphabetically sorted list of file names which match the pattern. In a list of words specifying filename substitution it is an error for no pattern to match an existing file name, but it is not required for each pattern to match. Only the metacharacters `*', `?' and `[' imply pattern matching, the characters `~' and `{' being more akin to abbreviations.

In matching filenames, the character `.' at the beginning of a filename or immediately following a `/', as well as the character `/' must be matched explicitly. The character `*' matches any string of characters, including the null string. The character `?' matches any single character. The sequence `[...]' matches any one of the characters enclosed. Within `[...]', a pair of characters separated by `-' matches any character lexically between the two.

The character `~' at the beginning of a filename is used to refer to home directories. Standing alone, i.e. `~' it expands to the invokers home directory as reflected in the value of the variable home. When followed by a name consisting of letters, digits and `-' characters the shell searches for a user with that name and substitutes their home directory; thus `~ken' might expand to `/usr/ken' and `~ken/chmach' to `/usr/ken/chmach'. If the character `~' is followed by a character other than a letter or `/' or appears not at the beginning of a word, it is left undisturbed.

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 `/usr/source'. Similarly `../{memo,*box}' might expand to `../memo ../box ../mbox'. (Note that `memo' was not sorted with the results of matching `*box'.) As a special case `{', `}' and `{}' are passed undisturbed.

Input/output

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

< name

Open file name (which is first variable, command and filename 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, filename 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 then it is created; if the file exists, its is truncated, its previous contents being lost.

If the variable noclobber is set, then the file must not exist or be a character special file (e.g. a terminal or `/dev/null') or an error results. This helps prevent accidental destruction of files. In this case the `!' forms can be used and suppress this check.

The forms involving `&' route the diagnostic output into the specified file as well as the standard output. Name is expanded in the same way as `<' input filenames are.

>> name
>>& name
>>! name
>>&! name

Uses file name as standard output like `>' but places output at the end of the file. If the variable noclobber is set, then it is an error for the file not to exist unless one of the `!' forms is given. Otherwise similar to `>'.

If a command is run detached (followed by `&') then the default standard input for the command is the empty file `/dev/null'. Otherwise the 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 run from a file of shell commands 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 command scripts to function as components of pipelines and allows the shell to block read its input.

Diagnostic output may be directed through a pipe with the standard output. Simply use the form `|&' rather than just `|'.

Expressions

A number of the builtin commands (to be described subsequently) take expressions, in which the operators are similar to those of C, with the same precedence. These expressions appear in the @, exit, if, and *while* commands. The following operators are available:

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

Here the precedence increases to the right, `==' and `!=', `<=' `>=' `<' and `>', `<<' and `>>', `+' and `-', `*' `/' and `%' being, in groups, at the same level. The `==' and `!=' operators compare their arguments as strings, all others operate on numbers. Strings which begin with `0' are considered octal numbers. Null or missing arguments are considered `0'. The result of all expressions are strings, which represent decimal numbers. It is important to note that no two components of an expression can appear in the same word; except when adjacent to components of expressions which are syntactically significant to the parser (`&' `|' `<' `>' `(' `)') they should be surrounded by spaces.

Also available in expressions as primitive operands are command executions enclosed in `{' and `}' and file enquiries of the form `-*l* name' 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 name is command and filename expanded and then tested to see if it has the specified relationship to the real user. If the file does not exist or is inaccessible then all enquiries return false, i.e. `0'. Command executions succeed, returning true, i.e. `1', if the command exits with status 0, otherwise they fail, returning false, i.e. `0'. If more detailed status information is required then the command should be executed outside of an expression and the variable *status* examined.

Control flow

The shell contains a number of commands which 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 in its input

and, due to the implementation, restrict the placement of some of the commands.

The *foreach*, *switch*, and *while* statements, as well as the *if-then-else* 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 up 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 goto's will succeed on non-seekable inputs.)

Builtin commands

Builtin commands are executed within the shell. If a builtin command occurs as any component of a pipeline except the last then it is executed in a subshell.

alias
alias name
alias name wordlist

The first form prints all aliases. The second form prints the alias for name. The final form assigns the specified *wordlist* as the alias of *name*; *wordlist* is command and filename substituted. *Name* is not allowed to be *alias* or *unalias*

alloc

Shows the amount of dynamic core in use, broken down into used and free core, and address of the last location in the heap. With an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the shell; it requires a modified version of the system memory allocator.

break

Causes execution to resume after the *end* of the nearest enclosing *forall* or *while*. 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 *switch*, resuming after the *endsw*.

case label:

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

cd

cd name
chdir
chdir name

Change the shells working directory to directory *name*. If no argument is given then change to the home directory of the user.

If *name* is not found as a subdirectory of the current directory (and does not begin with `/', `./', or `../'), then each component of the variable *cdpath* is checked to see if it has a subdirectory *name*. Finally, if all else fails but *name* is a shell variable whose value begins with `/', then this is tried to see if it is a directory.

continue

Continue execution of the nearest enclosing *while* or *foreach*. 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 *case* labels.

echo wordlist

The specified words are written to the shells standard output. A `\c' causes the echo to complete without printing a newline, akin to the `\c' in *nroff*(1). A `\n' in wordlist causes a newline to be printed. Otherwise the words are echoed, separated by spaces.

else
end
endif
endsw

See the description of the *foreach*, *if*, *switch*, and *while* statements below.

exec command

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

exit
exit(expr)

The shell exits either with the value of the *status* variable (first form) or with the value of the specified *expr* (second form).

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 builtin command *continue* may be used to continue the loop prematurely and the builtin command *break* to terminate it prematurely. When this command is read from the terminal, the loop is read up once prompting with `?' before any statements in the loop are executed. If you make a mistake typing in a loop at the terminal you can rub it out.

glob wordlist

Like *echo* but no `\' escapes are recognized and words are delimited by null characters in the output. Useful for programs which wish to use the shell to filename expand a list of words.

goto word

The specified *word* is filename and command expanded to yield a string of the form `label'. 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.

history

Displays the history event list.

if (expr) command

If the specified expression evaluates true, then 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. Input/output 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 the specified *expr* is true then the commands to the first *else* are executed; else if *expr2* is true then the commands to the second else are executed, etc. Any number of *else-if* pairs are possible; 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 *else*.)

login

Terminate a login shell, replacing it with an instance of

/bin/login. This is one way to log off, included for compatibility with **/bin/sh.**

logout

Terminate a login shell. Especially useful if *ignoreeof* is set.

nice
nice +number
nice command
nice +number command

The first form sets the *nice* for this shell to 4. The second form sets the *nice* to the given number. The final two forms run command at priority 4 and *number* respectively. The super-user may specify negative niceness by using `nice -number ...'. Command is always executed in a sub-shell, and the restrictions place on commands in simple *if* statements apply.

nohup
nohup command

The first form can be used in shell scripts to cause hangups to be ignored for the remainder of the script. The second form causes the specified command to be run with hangups ignored. On the Computer Center systems at UC Berkeley, this also *submits* the process. Unless the shell is running detached, *nohup* has no effect. All processes detached with ``&'' are automatically *nohup'ed*. (Thus, *nohup* is not really needed.)

onintr
onintr -
onintr label

Control the action of the shell on interrupts. The first form restores the default action of the shell on interrupts which is to terminate shell scripts or to return to the terminal command input level. The second form `onintr -' causes all interrupts to be ignored. The final form causes the shell to execute a `goto label' when an interrupt is received or a child process terminates because it was interrupted.

In any case, if the shell is running detached and interrupts are being ignored, all forms of *onintr* have no meaning and interrupts continue to be ignored by the shell and all invoked commands.

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 occurs exactly once, even if *count* is 0.

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

The first form of the command shows the value of all shell variables. Variables which have other than a single word as value 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 may 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

(Version 7 systems only.) Sets the value of environment variable *name* to be *value*, a single string. Useful environment variables are `TERM' the type of your terminal and `SHELL' the shell you are using.

shift
shift variable

The members of *argv* are shifted to the left, discarding *argv*[1]. It is an error for *argv* not to be set or to have less than one word as value. The second form performs the same function on the specified variable.

source name

The shell reads commands from *name*. *Source* commands may be nested; if they are nested too deeply the shell may run out of file descriptors. An error in a *source* at any level terminates all nested *source* commands. Input during *source* commands is **never** placed on the history list.

switch (string)
case str1:

...

breaksw

```
...
default:
    ...
  breaksw
endsw
```

Each case 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, then the execution begins after the default label. Each case label and the default label must appear at the beginning of a line. The command *breaksw* 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 *endsw*.

time
time command

With no argument, a summary of time used by this shell and its children is printed. If arguments are given the specified 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.

umask
umask value

The file creation mask is displayed (first form) or set to the specified value (second form). The mask is given in octal. Common values for the mask are 002 giving all access to the group and read and execute access to others or 022 giving all access except no write access for users in the group or others.

unalias pattern

All aliases whose names match the specified pattern are discarded. Thus all aliases are removed by `unalias *'. It is not an error for nothing to be *unaliased*.

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 `unset *'; this has noticeably distasteful side-effects. It is not an error for nothing to be *unset*.

wait

All child processes are waited for. It the shell is

interactive, then an interrupt can disrupt the wait, at which time the shell prints names and process numbers of all children known to be outstanding.

while (expr)
  ...
end

While the specified expression evaluates non-zero, the commands between the *while* and the matching end are evaluated. *Break* and *continue* may be used to terminate or continue the loop prematurely. (The *while* and *end* must appear alone on their input lines.) Prompting occurs here the first time through the loop as for the *foreach* statement if the input is a terminal.

@
@ name = expr
@ name[index] = expr

The first form prints the values of all the shell variables. The second form sets the specified *name* to the value of *expr*. If the expression contains `(', `)', `&' or `|' then at least this part of the expression must be placed within `(' `)'. The third form assigns the value of *expr* 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. The space separating the name from the assignment operator is optional. Spaces are, however, mandatory in separating components of *expr* which would otherwise be single words.

Special postfix `++' and `--' operators increment and decrement *name* respectively, i.e. `@ i++'.

Pre-defined variables

The following variables have special meaning to the shell. Of these, *argv*, *child*, *home*, *path*, *prompt*, *shell* and *status* are always set by the shell. Except for *child* and *status* this setting occurs only at initialization; these variables will not then be modified unless this is done explicitly by the user.

The shell copies the environment variable PATH into the variable *path*, and copies the value back into the environment whenever *path* is set. Thus is is not necessary to worry about its setting other than in the file *.cshrc* as inferior *csh* processes will import the definition of *path* from the environment. (It could be set once in the *.login* except that commands through *net*(1) would not see the definition.)

argv Set to the arguments to the shell, it is from this variable that positional parameters are

substituted, i.e. `$1' is replaced by `$argv[1]', etc.

cdpath
: Gives a list of alternate directories searched to find subdirectories in *chdir* commands.

child
: The process number printed when the last command was forked with `&'. This variable is *unset* when this process terminates.

echo
: Set when the -x command line option is given. Causes each command and its arguments to be echoed just before it is executed. For non-builtin commands all expansions occur before echoing. Builtin commands are echoed before command and filename substitution, since these substitutions are then done selectively.

histchars
: Can be assigned a two character string. The first character is used as a history character in place of ``!'', the second character is used in place of the ``^'' substitution mechanism. For example, ``set histchars=",;"'' will cause the history characters to be comma and semicolon.

history
: Can be given a numeric value to control the size of the history list. Any command which has been referenced in this many events will not be discarded. Too large values of *history* may run the shell out of memory. The last executed command is always saved on the history list.

home
: The home directory of the invoker, initialized from the environment. The filename expansion of `~' refers to this variable.

ignoreeof
: If set the shell ignores end-of-file from input devices which are terminals. This prevents shells from accidentally being killed by control-D's.

mail
: The files where the shell checks for mail. This is done after each command completion which will result in a prompt, if a specified interval has elapsed. The shell says `You have new mail.' if the file exists with an access time not greater than its modify time.

: If the first word of the value of *mail* is numeric it specifies a different mail checking interval, in seconds, than the default, which is 10 minutes.

: If multiple mail files are specified, then the

shell says `New mail in *name*' when there is mail in the file *name*.

noclobber
As described in the section on *Input/output*, restrictions are placed on output redirection to insure that files are not accidentally destroyed, and that `>>' redirections refer to existing files.

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

nonomatch
If set, it is not an error for a filename expansion to not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, i.e. `echo [' still gives an error.

path
Each word of the path variable specifies a directory in which commands are to be sought for execution. A null word specifies the current directory. If there is no *path* variable then only full path names will execute. The usual search path is `.', `/bin' and `/usr/bin', but this may vary from system to system. For the super-user the default search path is `/etc', `/bin' and `/usr/bin'. A shell which is given neither the **-c** nor the **-t** option will normally hash the contents of the directories in the *path* variable after reading *.cshrc*, and each time the *path* variable is reset. If new commands are added to these directories while the shell is active, it may be necessary to give the *rehash* or the commands may not be found.

prompt
The string which is printed before each command is read from an interactive terminal input. If a `!' appears in the string it will be replaced by the current event number unless a preceding `\' is given. Default is `% ', or `# ' for the super-user.

shell
The file in which the shell resides. This is used in forking shells to interpret files which have execute bits set, but which are not executable by the system. (See the description of *Non-builtin Command Execution* below.) Initialized to the (system-dependent) home of the shell.

status
The status returned by the last command. If it

terminated abnormally, then 0200 is added to the status. Builtin commands which fail return exit status `1', all other builtin commands set status `0'.

time Controls automatic timing of commands. If set, then any command which takes more than this many cpu seconds will cause a line giving user, system, and real times and a utilization percentage which is the ratio of user plus system times to real time to be printed when it terminates.

verbose Set by the **-v** command line option, causes the words of each command to be printed after history substitution.

Non-builtin command execution

When a command to be executed is found to not be a builtin command the shell attempts to execute the command via *exec*(2). Each word in the variable *path* names a directory from which the shell will attempt to execute the command. If it is given neither a **-c** nor a **-t** option, the shell will hash the names in these directories into an internal table so that it will only try an *exec* in a directory if there is a possibility that the command resides there. 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 *unhash*), or if the shell was given a **-c** or **-t** argument, and in any case for each directory component of *path* which does not begin with a ``/'', the shell concatenates with the given command name to form a path name of a file which it then attempts to execute.

Parenthesized commands are always executed in a subshell. Thus `(cd ; pwd) ; pwd' prints the *home* directory; leaving you where you were (printing this after the home directory), while `cd ; pwd' leaves you in the *home* directory. Parenthesized commands are most often used to prevent *chdir* from affecting the current shell.

If the file has execute permissions but is not an executable binary to the system, then it is assumed to be a file containing shell commands an a new shell is spawned to read it.

If there is an *alias* for *shell* then the words of the alias will be prepended to the argument list to form the shell command. The first word of the *alias* should be the full path name of the shell (e.g. `$shell'). Note that this is a special, late occurring, case of *alias* substitution, and only allows words to be prepended to the argument list without modification.

Argument list processing

If argument 0 to the shell is `-' then this is a login shell. The flag arguments are interpreted as follows:

-c Commands are read from the (single) following argument which must be present. Any remaining arguments are placed in *argv*.

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

-f The shell will start faster, because it will neither search for nor execute commands from the file `.cshrc' in the invokers home directory.

-i The shell is interactive and prompts for its top-level input, even if it appears to not be a terminal. Shells are interactive without this option if their inputs and outputs are terminals.

-n Commands are parsed, but not executed. This may aid in syntactic checking of shell scripts.

-s Command input is taken from the standard input.

-t A single line of input is read and executed. A `\' may be used to escape the newline at the end of this line and continue onto another line.

-v Causes the *verbose* variable to be set, with the effect that command input is echoed after history substitution.

-x Causes the *echo* variable to be set, so that commands are echoed immediately before execution.

-V Causes the *verbose* variable to be set even before `.cshrc' is executed.

-X Is to **-x** as **-V** is to **-v**.

After processing of flag arguments if arguments remain but none of the **-c**, **-i**, **-s**, or **-t** options was given the first argument is taken as the name of a file of commands to be executed. The shell opens this file, and saves its name for possible resubstitution by `$0'. Since many systems use either the standard version 6 or version 7 shells whose shell scripts are not compatible with this shell, the shell will execute such a `standard' shell if the first character of a script is not a `#', i.e. if the script does not start with a comment. Remaining arguments initialize the variable *argv*.

Signal handling

The shell normally ignores quit signals. The interrupt and quit signals are ignored for an invoked command if the command is followed by `&'; otherwise the signals have the values which the shell inherited from its parent. The shells handling of interrupts 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 `.logout'.

AUTHOR

William Joy

FILES

~/.cshrc	Read at beginning of execution by each shell.
~/.login	Read by login shell, after `.cshrc' at login.
~/.logout	Read by login shell, at logout.
/bin/sh	Standard shell, for shell scripts not starting with a `#
/tmp/sh*	Temporary file for `<<'.
/dev/null	Source of empty file.
/etc/passwd	Source of home directories for `~name'.

LIMITATIONS

Words can be no longer than 512 characters. The number of characters in an argument varies from system to system. Early version 6 systems typically have 512 character limits while later version 6 and version 7 systems have 5120 character limits. The number of arguments to a command which involves filename expansion is limited to 1/6'th the number of characters allowed in an argument list. Also command substitutions may substitute no more characters than are allowed in an argument list.

To detect looping, the shell restricts the number of alias substititutions on a single line to 20.

SEE ALSO

access(2), exec(2), fork(2), pipe(2), signal(2), umask(2), wait(2), a.out(5), environ(5), `An introduction to the C shell'

BUGS

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.

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

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.

Some commands should not touch status or it may be so transient

as to be almost useless. Oring in 0200 to *status* on abnormal termination is a kludge.

In order to be able to recover from failing *exec* commands on version 6 systems, the new command inherits several open files other than the normal standard input and output and diagnostic output. If the input and output are redirected and the new command does not close these files, some files may be held open unnecessarily.

There are a number of bugs associated with the importing/exporting of the PATH. For example, directories in the path using the ~ syntax are not expanded in the PATH. Unusual paths, such as (), can cause csh to core dump.

This version of *csh* does not support or use the process control features of the 4th Berkeley Distribution. It contains a number of known bugs which have been fixed in the process control version. This version is not supported.