NAME

lsh − the Layla shell

SYNOPSIS

lsh [**−abBCdeEfghiklLmnpqrtTuvwxy**] [**−o** *opt*] [**−O** *extopt*] [*arg ...*]
lsh [**−abBCdeEfghiklLmnpqrtTuvwxy**] [**−o** *opt*] [**−O** *extopt*] **-c** *string* [*arg ...*]
lsh [**−abBCdeEfghiklLmnpqrtTuvwxy**] [**−o** *opt*] [**−O** *extopt*] [**−is**] [*arg ...*]

DESCRIPTION

Layla shell is a POSIX-compliant GNU/Linux shell. It aims to implement the full functionality specified in the POSIX Shell & Utilities volume. This volume describes the Shell Command Language, the general terminal interface, and the builtin utilities of compliant shells. The full POSIX standard is freely available from The Open Group's website at:

https://pubs.opengroup.org/onlinepubs/9699919799.2018edition/

Layla shell implements most of the POSIX-defined functionality. Specifically, quoting, token recognition, words expansions, I/O redirection, reponses to error conditions, the shell command language, the standard shell execution environment, signal handling, and the general and special builtin utilities are all implemented and behave in a way that is conforming to the POSIX standard. Pattern matching and pathname expansion relies on external tools, and thus might not behave exactly as specified in the POSIX standard. The shell grammar has been extended to accommodate some of the widely used non- POSIX words, such as the function and [[keywords and the (( )) arithmetic expansion operator. In all of these situations, we followed ksh behaviour, as ksh is the "model" POSIX shell, the one upon which the Shell & Utilities volume was modeled in the first place. But ksh is not the only shell out there. This is why we included features from other shells, most notably bash, the most widely used shell nowadays.

To better understand the inner workings of this (as well as any other POSIX- compliant shell), it would be better if you got a copy of the Linux Shell Internals book, published by No Starch Press. This shell and the book has been developed hand-in-hand. The book helps to explain the code of this shell and provide a walk-through for newcomers. Although the code is extensively commented, a lot of theoretical ground has covered in the book, not in the source code. If you really want to understand how and why Unix/Linux shells behave in the context of POSIX, I honestly advise you to get a copy of the book.

There is still a long way to go with testing, bug-fixing and improving Layla shell, and your feedback is more than welcome in this regard. If you have a bug report, or you want to suggest adding some feature or fixing something with the shell, feel free to email me directly at my email address, the one you will find in the beginning of this file. If you have any bugfixes, patches or feature improvements you want to add to the code, feel free to send me your code at the email address above, or through the Linux Shell Internals repository at GitHub:

https://github.com/moisam/Linux-Shell-Internals-book/

OPTIONS

−− End of options.

−−help Print command line help and exit.

−−init-file, **−−rcfile**
Specify the startup script file to read instead of the default *~/.lshrc*.

−−login, **−l**
Start a login shell (as if called by the login utility).

−−noprofile
Do not read login startup scripts */etc/profile* and *~/.profile*.

−−norc Do not read the startup script files */etc/lshrc* and *~/.lshrc*.

−−restricted
Start the shell in the restricted mode.

−−verbose
Turn on the **-v** option, so commands are printed as they are read.

--version
Display shell version and exit.

−o *opt*, **+o** *opt*
Turn on '-' or off '+' shell options, as if calling the **set** builtin utility.

−O *extopt*, **+O** *extopt*
Turn on '-' or off '+' extended shell options, as if calling the *setx* (or the *shopt*) builtin utility.

INSTALLATION

Package dependencies:

Layla shell has very few external dependencies, in order to ease the process of compiling and installing it. You only need to have the GNU C library installed, in addition to the GNU Compiler Collection (GCC) or the Clang/LLVM compiler.

If you are using any GNU/Linux distro, it would be better if you checked your distro's official repositories, as these tools are installed by default on most systems. If, by a sore chance, you needed to manually download and install them, here are the links:
- GNU C Library: https://www.gnu.org/software/libc/
- GCC: https://gcc.gnu.org/
- LLVM: http://releases.llvm.org/download.html

How to compile and install:

If you downloaded the shell as a source tarball, navigate to the directory where you downloaded the tarball:

```
$ cd ~/Downloads
```

then extract the archive and enter the extracted directory:

```
$ tar xvf lsh-1.0
$ cd lsh-1.0/
```

then run:

```
$ make && sudo make install
```

and that's it! Now you can run the shell by invoking:

```
$ lsh
```

COMMANDS

Commands can be either simple or compound commands, and can be grouped together to form pipelines and lists. Every command that executes returns an exit status to the shell. An exit status of zero is considered success (or true, when used as a value for logical testing). A non-zero exit status is considered failure (or false).

A simple command is a list of variable assignments and words separated by whitespace. The first word usually specifies the name of the command to be executed, the remaining words are word-expanded and passed as arguments to the command. The command name is passed as argument number 0. If the command exits normally, the exit status is 0-128. If it terminates abnormally, the exit status is 128+signum (similar to bash, but not ksh). The name of the signal can be obtained by passing the –l option to the kill builtin utility.

A pipeline is a sequence of one or more commands separated by the pipe operator '|'. The standard output (stdout) of each command except the last is connected by a pipe to the standard input (stdin) of the next command. Each command is run in a separate process. The shell waits for all commands to terminate. The exit status of a pipeline is the exit status of the last command unless the pipefail '-l' option is enabled, in which case it is the exit status of the rightmost command with non-zero exit status. Each pipeline can be preceded by the bang reserved word '!'. This causes the exit status of the pipeline to be inverted, i.e. an exit status of zero becomes 1, otherwise it becomes zero.

A list is a sequence of one or more pipelines separated by ';', '&', '|&', '&&', or '|'. The ';' causes

sequential (foreground) execution of the preceding pipeline. The shell waits for foreground pipelines to finish execution. The '&' causes asynchronous (background) execution of the preceding pipeline. The shell does not wait for background pipelines to finish execution. The '|&' causes asynchronous execution of the preceding pipeline, redirecting both stdout and stderr to the pipe. This is similar to bash behavior, while ksh uses this operator to establish a two-way pipe between the command and the parent shell. The symbols '&&' and '||' cause the list following it to be executed only if the preceding pipeline returns a zero or non-zero value, respectively. Semicolons ';' can be substituted by newlines in lists.

Compound commands include **for**, **select**, **until** and **while** loops; **case** and **if** conditionals; subshells and brace groups. The exit status of a compound command is that of the last simple command executed in the compound. All loops conform to the POSIX standard, except when it says otherwise.

for name [in word ...]; do list; done

Each time the **for** loop is executed, **name** is set to the next word taken from the **word** list. If the word list is omitted, the for loop behaves as if **in "$@"** was provided (which loops on positional parameters, starting from 1). The loop exits when there are no more words in the list.

(([expr1] ; [expr2] ; [expr3])); do list; done

The arithmetic expression **expr1** is evaluated. Arithmetic expression **expr2** is then evaluated repeatedly until it evaluates to zero. Every time **expr2** evaluates to non-zero, **list** is executed and arithmetic expression **expr3** is evaluated. If any expression is omitted, it behaves as if it evaluated to 1. This loop is a non-POSIX extension.

select name [in word ...]; do list; done

The **select** loop prints the set of **word**s on stderr, each preceded by its numeric index. If the list of **word**s is omitted, the loop behaves as if **in "$@"** was provided (which loops on positional parameters, starting from 1). The **PS3** prompt is printed and a single line is read from stdin. If the read line consists of the number of one of the listed words, the value of variable **name** is set to the **word** corresponding to this number. If the read line is empty, the selection list is printed again. In all other cases, variable **name** is set to null. The contents of the read line is saved in the shell variable **$REPLY**. The list is executed repeatedly until a break or EOF is encountered. If the **$REPLY** variable is set to null as a side effect of executing **list**, the selection list is printed again before displaying the PS3 prompt and waiting for the next selection. This loop is a non-POSIX extension used by bash and ksh.

case word in [[(] pattern [| pattern] ...) list ;;] ... esac

The **case** conditional executes the **list** associated with the first **pattern** that matches the given **word**. The form of the patterns is the same as that used for file name generation (i.e. can contain the wildcards *, ? and []). The ';;' operator causes execution of the **case** to terminate. If ';&' is used instead, the next subsequent list, if any, is executed. If ';;&' is used, the shell tries to match and execute another **case** in the list. Both ';&' and ';;&' are non-POSIX extensions used by shells like ksh and bash.

if list1; then cmd-list1 [; elif list2; then cmd-list2] ... [; else cmd-list3]; fi

list1 is executed and if it returns zero exit status, **cmd-list1** is executed. Otherwise, **list2** is executed and if its value is zero, **cmd-list2** is executed. If all **elif** lists fail, the **else**'s **cmd-list3** is executed. If **list1** has non-zero exit status and there is no **else** clause, the **if** command returns zero exit status.

while conditional-list; do cmd-list; done
until conditional-list; do cmd-list; done

The **while** loop repeatedly executes the **conditional-list** and if the exit status of the last command in the list is zero, it executes **cmd-list**, otherwise the loop terminates. If no commands are executed from the **cmd-list**, the **while** loop returns a zero exit status. **until** can be used instead of **while** to negate the loop termination test (i.e. execute until **conditional-list** has zero exit status).

((expression))

expression is evaluated as an arithmetic expression. If the value is non-zero, the exit status is 0. Otherwise the exit status is 1. This is a non- POSIX extension.

(list;)

Execute **list** in a separate environment, i.e. a subshell. If two adjacent open parentheses are needed (for nesting subshells), a space must be inserted between the two open parentheses to avoid evaluation as an arithmetic expression. '{' and '}' are reserved words; they must occur at the beginning of a line (after a newline character) or after a ';' to be recognized.

[[expression]]

Evaluate **expression** and return a zero exit status if **expression** evalues to true (zero exit status). See *Conditional Expressions* for a description of what **expression** can be.

function funcname { list ; }
funcname () { list ; }

Define a function named *funcname*. The body of the function is the list of commands between '{' and '}'. The body is executed when the function is called, not when its defined. The exit status of the function definition command is zero, unless an error occurs in adding the function to the symbol table. The exit status of a function call is that of the last command executed in the function, or the value passed to the **return** builtin. The first syntax using the **function** keyword is a non-POSIX extension used by bash and ksh.

time [pipeline]

If **pipeline** is omitted, the *user* and *system* time for the current shell and completed child processes is printed on stderr. Otherwise, the **pipeline** is executed and the elapsed *real* time, as well as the *user* and *system* times are printed on stderr. The **$TIMEFORMAT** variable controls how the timing information should be displayed (See the *Shell Variables* section for a description of **$TIMEFORMAT**).

RESERVED WORDS

The following reserved words are recognized as reserved only when they are the first unquoted word of a command:

case
do
done
else
elif
esac
for
fi
function
if
select
then
until
while
{ }
!

All reserved words (or keywords) are defined by POSIX, except **function** and **select**.

VARIABLE ASSIGNMENTS

Simple commands can start with one or more variable assignments. Variable assignments occurring after command names are recognized as such if the **yword** '**−k**' option is set. Assignments can also be passed as arguments to the **export** or **readonly** special builtin utilities. The syntax for an assignment takes the form of:

varname=value
No space is permitted between **varname** and the '=', or between the '=' and **value**.
Layla shell currently does not support the '+=' operator for variable assignents.

COMMENTS

The '#' character causes all the following characters up to the next newline character to be commented, or ignored. Interactive shells recognize comment characters when the extended option **interactive-comments** (or **interactive_comments**) is set by calling the **setx** builtin (which behaves similar to bash's **shopt**).

ALIASING

The first word of each simple command is replaced by the text of an alias if an alias for the command word has been defined. An alias name consists of any number of alphanumeric characters and any of '_', '!', '%', ',' and '@'. The replacement string can contain any valid shell command. The first word of each command in the replaced text is tested for aliases, except when the command is the same one being aliased (to avoid infinite looping). If the last character of the alias value is a space character, the word following the alias is also checked for alias substitution.

Aliases cannot be used to redefine shell's reserved words. Aliases can be created and listed using the **alias** command. They can be removed with the **unalias** command.

Aliasing is performed when scripts are read, not while they are executed. The following aliases are defined by default by the shell, but can be unset or redefined if the user wishes:

```
ls="ls --color=auto"
ll="ls -la"
l.="ls -d .* --color=auto"
cd..="cd .."
".."="cd .."
"..."="cd ../../../"
grep="grep --color=auto"
egrep="egrep --color=auto"
fgrep="fgrep --color=auto"
bc="bc -l"
vi="vim"
command="command "
nohup="nohup "
stop="kill -s STOP"
suspend="kill -s STOP $$"
hist="fc"
typeset="declare"
shopt="setx"
reboot="sudo /sbin/reboot"
poweroff="sudo /sbin/poweroff"
halt="sudo /sbin/halt"
shutdown="sudo /sbin/shutdown"
df="df -H"
du="du -ch"
r="fc -s"
memuse="memusage"
```

Some of the above aliases are defined by shells like bash and ksh, while others are lsh-specific, included for convenience.

SPECIAL ALIASES

The tcsh shell has a useful feature where a special group of aliases contain commands which the shell executes at certain times. For example, the @code{beepcmd} special alias can be defined to contain a command line the shell executes when it wants to ring the bell. As this feature can be very handy to shell users, we include it here, despite the fact it is not described by POSIX.

The following table describes the special aliases and their uses in lsh.

beepcmd
command to be run when the shell wants to ring the bell

jobcmd
command to be run before executing commands and when a job changes its state

periodic
command to be run every **TPERIOD** minutes

precmd
command to be run before printing the next command prompt

postcmd
command to be run before executing commands

TILDE EXPANSION

After alias substitution is performed, each word (or field) is checked to see if it begins with an unquoted tilde '~'. If it does, the word up to the first unquoted '/' is checked to see if it matches a user name in the password database (this word is known as the *tilde prefix*). If a match is found, the '~' and the matched user name (the *tilde prefix*) are replaced by the login (or home) directory of the matched user (POSIX). If no match is found, the original text is left unchanged. A '~' by itself, or in front of a '/' with no intervening characters, is replaced by the value of **$HOME** (POSIX). A '~' followed by a '+' or '-' is replaced by the value of **$PWD** and **$OLDPWD**, respectively (non-POSIX extension used by ksh and bash). A '~' followed by '~+N', '~-N' or '~N' is replaced by the corresponding directory stack entry, as if by calling the **dirs** builtin with '+N', '-N' or 'N' (non-POSIX bash extension).

Tilde expansion is also attempted during variable assignment if the value of the assignment begins with a '~', and when a '~' appears after a colon ':'.

COMMAND SUBSTITUTION

Command substitution occurs when commands are enclosed in parentheses preceded by a dollar sign '$', or when enclosed in a pair of back-quotes or grave accents '`'. In the back-quoted form, the string between quotes is processed for quoting characters before the command is executed.

The special command substitution *$(cat file)* can be replaced by the equivalent but faster *$(<file)* (non-POSIX ksh and bash extension). The command substitution *$(n<#)* expands to the current byte offset for file descriptor *n* (non-POSIX ksh extension).

ARITHMETIC EXPANSION

Arithmetic expressions are enclosed in double parentheses preceded by a dollar sign, in the form of:

$((arithmetic-expression))

Arithmetic expressions can also be written without the preceding dollar sign (non-POSIX extension):

((arithmetic-expression))

Arithmetic expressions are processed and replaced by the value of the expression inside the parentheses.

Arithmetic valuations are performed using long integer arithmetic, the only format required by POSIX, although other shells (e.g. bash and ksh) perform double or long double precision arithmetic. Numeric constants can be of the form *[base#]n*, where *base* is a decimal number between 2 and 36 (ksh and bash allow bases up to 64) representing the arithmetic base. *n* is a number in the given base. Digits greater than 9 are represented by lowercase or uppercase letters.

Only the basic arithmetic operations are currently supported in Layla shell. Specifically, floating point arithmetic and braced expressions are not supported. Also, the use of mathmetical functions (including **pow()** for exponentiation) is not supported.

PROCESS SUBSTITUTION

ksh, bash and other shells support process substitution on some UNIX operating systems that support the */dev/fd* directory scheme for naming open files. Layla shell currently doesn't support process substitution.

PARAMETER EXPANSION

A parameter can be a variable name, one or more digits, or any of the special characters '*', '@', '#', '?', '-', '$', and '!'. A variable is denoted by a name, which consists of alphanumeric characters and '_', and must begin with an alphabetic character or '_'. Exported variables are passed in the environment to child processes. Layla shell currently doesn't support indexed or associative arrays, or nameref variables.

The value of a variable can be assigned by:

name=value [name=value] ...

No space is allowed before or after the '='. Positional parameters cannot be assigned in this way; they must be set using the **set** or **shift** special builtin utilities. Parameter **$0** is set from argument zero (**argv[0]**) when the shell is invoked. The character '$' is used to indicate parameter expansion, which can be of the following formats:

${parameter}

The parameter name contains all characters from '${' to the matching '}'. The parameter's value, if any, is substituted. The braces are required when **parameter** is followed by a letter, digit, or underscore that is not part of the parameter name. If **parameter** is one or more digits this indicates a *positional parameter*. A positional parameter of more than one digit must be enclosed in braces. If **parameter** is '*' or '@', all the positional parameters (starting with **$1**) are substituted and separated by a field separator character.

${#parameter}

If **parameter** is '*' or '@', the number of positional parameters is substituted. Otherwise, the length of **parameter**'s value is substituted.

${!prefix*}

Expands to the names of the variables whose names begin with **prefix** (non- POSIX extension).

${parameter:-word}

If **parameter** is set and is non-null, its value is substituted. Otherwise **word** is substituted. **word** is not evaluated unless it is to be used as the substituted string. If the colon is omitted, the shell only checks whether parameter is set or not.

${parameter:offset:length}
${parameter:offset}

Expands to the portion of the value of **parameter** starting at the character number resulting from the expansion of the **offset** arithmetic expression (counting from zero), and consisting of the number of characters resulting form the expansion of the **length** arithmetic expression. In the second form, all chacaters till the end of the value are used. A negative offset counts backwards from the end, and one or more spaces are required before minus sign to prevent the shell from interpreting the operator as ':-'. If **parameter** is '*' or '@', **offset** and **length** refer to the array index and number of elements respectively. A negative offset is taken relative to the number of elements in the array. This format is a non-POSIX extension.

${parameter#pattern}
${parameter##pattern}

If **pattern** matches the beginning (prefix) of the value of **parameter**, the value substituted is the value of **parameter** with the matched portion deleted. Otherwise the value of **parameter** is substituted. In the first form the shortest matching pattern is deleted and in the second form the longest matching pattern is deleted (POSIX). When **parameter** is '@' or '*', the substring operation is applied to each element in turn (non-POSIX ksh and bash extension).

${parameter%pattern}
${parameter%%pattern}

If **pattern** matches the ending (suffix) of the value of **parameter**, the value substituted is the value of **parameter** with the matched portion deleted. Otherwise the value of **parameter** is substituted. In the first form the shortest matching pattern is deleted and in the second form the longest matching pattern is deleted (POSIX). When **parameter** is '@' or '*', the substring operation is applied to each element in turn (non-

POSIX ksh and bash extension).

${parameter/pattern/string}
${parameter//pattern/string}
${parameter/#pattern/string}
${parameter/%pattern/string}

These are non-POSIX extensions and are not currently implemented in the Layla shell.

FIELD SPLITTING

After parameter expansion and command substitution, the results are scanned for the field separator characters (as indicated by **$IFS**) and split into separate fields. Explicit null fields (those passed as "" or '') are retained. Implicit null fields resulting from parameters with no values or command substitutions with no output are removed.

If the **braceexpand** or '**–B**' option is set, the fields are checked for brace patterns, such as *{*,*}*, *{l1..l2}*, and *{n1..n2}*, where '*' means any character; 'l1' and 'l2' are letters and 'n1' and 'n2' are signed numbers. Fields are generated by prepending the characters before the opening brace and appending the characters after the closing brace to each of the strings generated by the characters between the braces.

PATHNAME EXPANSION

(also known as File Name Generation)

Following field splitting, each field is scanned for the regular expression (regex) characters '*', '?', and '[' (unless the **noglob** or '**–f**' option is set. If the field contains a regex character, it is treated as a pattern.

Pathnames containing pattern characters are replaced with sets of names that matches the pattern. If no pathname is found that matches the pattern, that pathname is left unchanged. If **$FIGNORE** is set, each pathname that matches the pattern defined by the value of **$FIGNORE** is ignored when generating the list. The names '.' and '..' are also ignored.

The following is what the pattern characters mean inside pattern strings:

***** Match any string including the null string.

? Match any single character.

[...] Match any one of the enclosed characters. Two characters separated by '-' matches any character lexically between the pair, inclusive. If the first character is a bang '!', any character not enclosed is matched. A '-' can be included in the character set by putting it as the first or last character. Character classes can be specified with the syntax [:class:] where class is one of the following ANSI-C classes: **alnum alpha blank cntrl digit graph lower print punct space upper word xdigit**.

QUOTING

Quoting can be used to remove the special meaning of the shell's metacharacters, which include:

; & ( ) | < > newline space tab

Metacharacters are used to delimit tokens, in addition to their special meaning to the shell. If a character is quoted, that is, preceded with a backslash '\', it is quoted and loses its special meaning to the shell. The pair \newline is removed from input. All characters enclosed between single quotes are considered to be quoted. Single quotes cannot appear within single quotes. A single quoted string preceded by an unquoted dollar sign '$' is processed as an ANSI-C string. Parameter expansion and command substitution does not occur inside single quotes.

Inside double quotes, parameter and command substitution occur and '\' quotes the characters '\', '´', '`', '"', and '$'. If a '$' is found in front of a double quoted string, ksh and bash translate the string by a locale-specific message, but our shell currently ignores the dollar sign.

The shell variables '$*' and '$@' are identical when not quoted. However, "$*" is equivalent to **"$1c$2c..."**, where **c** is the first character of the **$IFS** variable, whereas "$@" is equivalent to "$1" "$2".

Inside back-quotes (grave accents), '\' quotes the characters '\', '`', and '$'.

The special meaning of reserved words (keywords) and aliases can be removed by quoting any character of the reserved word or alias.

SPECIAL PARAMETERS

The following special parameters are automatically set by the shell. They are all defined by POSIX, except when it says otherwise.

@ The positional parameters, starting from 1, producing separate fields for each parameter If expansion occurs inside double quotes, each parameter expands to a separate field

***** The positional parameters, starting from 1, producing separate fields for each parameter If expansion occurs inside double quotes, parameters are joined to form a single field

The decimal number of positional parameters, not counting parameter 0 or **$0**.

- Options supplied to the shell on invocation or by calling the **set** command.

? The decimal value returned by the last executed command.

$ The process identifier (**PID**) of this shell.

_ Initially, the value of '_' is the absolute pathname of the shell or script being executed (as passed in the environment at startup). It is subsequently assigned the last argument of the previous command (bash and ksh, while csh assigns the full command line of the last command). This parameter is not set for asynchronous (background) commands. It also holds the name of the matching **$MAIL** file when checking for mail. This parameter is a non-POSIX extension.

! The **PID** of the last run background pipeline, or the most recent job put in the background with the **bg** builtin utility.

0 The name of this shell or script.

SHELL VARIABLES

Shell variables are initialized from the environment at startup. During the shell's lifetime, shell variables can have their values reassigned or unset through variable assignment. The following are the variables used by the shell. They are all defined by POSIX, except when it says otherwise.

COMMAND

Similar to bash's **$BASH_COMMAND** variable, this variable contains the full command line of the currently executing command. This variable is a non-POSIX extension.

COMMAND_STRING

Similar to bash's **$BASH_EXECUTION_STRING** variable, this variable contains the value of the command string passed to the shell with the **−c** option. This variable is a non-POSIX extension.

CDPATH

This variable efines the search path for the **cd** builtin.

COLUMNS

This variable defines the width of the shell's window for printing **select** lists and the results of tab completions.

COPROC0

The file descriptor of the file holding the coprocess's output, created by the **coproc** builtin utility. This variable is similar to bash's **$COPROC[0]**, which is an array variable.

COPROC1

The file descriptor of the file holding the coprocess's input, created by the **coproc** builtin utility. This variable is similar to bash's **$COPROC[1]**, which is an array variable.

DIRSFILE

Similar to tcsh's **$dirsfile** variable, this variable contains the pathname of the file used to save and restore the directory stack. This variable is a non-POSIX extension. See also **$SAVEDIRS**.

DIRSTACK
The contents of the directory stack in the order displayed by the **dirs** builtin. Assignments to this variable change the contents of the directory stack. This variable is a non-POSIX bash extension. It is similar to tcsh's **dirstack** variable.

EDITOR
If the **$VISUAL** variable is not set, the value of **$EDITOR** is processed and the corresponding editing mode is turned on (only the **vi** editing mode is currently supported). This variable is also used by the **fc** builtin to determine the editor program to use for editing history commands. This variable is a non-POSIX extension. The default value is **vi**.

EGID The effective group id of the current user. This variable is a non-POSIX extension.

ENV Parameter expansion, command substitution, and arithmetic expansion are performed on the value of this variable to generate the pathname of the script that is executed when the shell is invoked. This file is typically used for alias and function definitions. The default value is *$HOME/.lshrc*.

EPOCHREALTIME
The number of seconds (in floating point format) since Unix epoch. This variable is a non-POSIX bash extension.

EPOCHSECONDS
The number of seconds (in long integer format) since Unix epoch. This variable is a non-POSIX bash extension.

EUID The effective user id of the current user. This variable is a non-POSIX bash extension.

EXECIGNORE
This is a colon-separated list of patterns of executable file names to ignore when the shell is searching for executable files. This variable is a non- POSIX bash extension.

FCEDIT
The default editor name for the **fc** command. **$FCEDIT** takes precedence over **$HISTEDIT**, which in turn takes precedence over **$EDITOR**. The default value for **$FCEDIT** is **vi**.

FIGNORE
A pattern that defines the set of file names that is ignored when performing file name matching. This variable is a non-POSIX ksh extension.

FUNCNAME
The name of the currently executing function. The value is **'main'** if there is no function currently executing. This variable is similar to bash's **$FUNCNAME**, except that the latter is an array variable containing the names of all functions in the call stack. This variable is a non-POSIX bash extension.

FUNCNEST
The maximum function nesting level. Function calls cannot exceed this value, if it is set. This variable is a non-POSIX bash extension.

GID The numeric group id of the currently logged user. This variable is a non-POSIX extension.

GLOBIGNORE
A colon-separated list of patterns that define the set of file names to be ignored when performing file name (i.e. pathname) expansion. This variable is a non-POSIX bash extension.

GROUPS
The list of groups of which the current user is a member. This variable is similar to bash's **$GROUPS**, except that the latter is an array variable containing the group names. This variable is a non-POSIX bash extension.

HISTCMD
The number of the current command in the history file. This variable is a non-POSIX extension used by ksh and bash.

HISTCONTROL
A colon-separated list that controls how history entries are saved in the history file. The values of this list can be:

ignorespace	Don't save lines starting with a space.
ignoredups	Don't save lines matching the previous history entry.
ignoreboth	Shorthand for **ignorespace** and **ignoredups**.
erasedups	Remove duplicate lines matching the current one.

This variable is a non-POSIX bash extension.

HISTFILE
The pathname of the file that is used to store the command history.

HISTFILESIZE
The maximum number of commands to save in the history file. This variable is a non-POSIX bash extension.

HISTIGNORE
A colon-separated list that tells the shell which commands to ignore when saving the history list. Patterns are matched against the history line to be saved. If they match, the line is not saved in the history list. This variable is a non-POSIX bash extension.

HISTSIZE
The number of previously entered commands that are accessible by this shell. The default is 512.

HISTTIMEFORMAT
If set and not null, the value of this variable is the format string passed to **strftime()** in order to print the timestamp of history entries, which is output by the builtin **history** utility. This variable is a non-POSIX bash extension.

HISTEDIT
The name for the default editor name for the **fc** command. This variable is a non-POSIX ksh extension. It takes precedence over **$EDITOR**.

HOME
The default argument for the **cd** builtin (the home directory). The value of **$HOME** is typically set by **login**.

HOST

HOSTNAME
The name of the current host. Both variables are non-POSIX extensions.

HOSTFILE
The name of the file the shell reads when performing hostname completion. If this variable is null or not set, the default file **/etc/hosts** is used. This variable is a non-POSIX bash extension.

HOSTTYPE
The type of machine the shell is running on, determined at compilation time. Similar to **$MACHTYPE**. This variable is a non-POSIX bash extension.

IGNOREEOF
If set, indicates the count of consecutive EOFs the shell must read before exiting. If not set or null, the default is 10 (same as bash). This variable is only of use to interactive shell. This variable is a non-POSIX bash extension.

IFS Internal field separators, which default to space, tab, and newline. Used to separate the results of command substitution or parameter expansion and to separate fields when using the **read** builtin utility. The first character of the **$IFS** variable is used to separate arguments for the "$*" substitution. See the *Quoting* section. The default value is " \t\n".

INSERTMODE
The insert mode for the command line editor. If set to **overwrite**, the editor overwrites characters in the command line buffer (as if the **INSERT** key was pressed. Any other value, including **insert**, puts the editor in the regular mode, where characters are added without overwriting existing characters. This is a non-POSIX extension that behaves like tcsh's **inputmode** variable.

LANG The locale category for any category not specifically selected with a variable starting with LC_ or LANG. Not currently used by the shell.

LC_ALL
Overrides the value of the LANG variable and any other LC_ variable. Not currently used by the shell.

LC_COLLATE
The locale category for character collation information. Not currently used by the shell.

LC_CTYPE
The locale category for character handling functions. Not currently used by the shell.

LC_NUMERIC
The locale category for the decimal point character. Not currently used by the shell.

LINES The number of lines in the shell's window.

LINENO
The current line number within the script or function being executed.

MACHTYPE
The type of machine the shell is running on, determined at compilation time. Similar to **$HOSTTYPE**. This variable is a non-POSIX extension.

MAIL The name of the mail file to check if **$MAILPATH** is not set. **$MAIL** is not set by the shell.

MAILCHECK
The interval (in seconds) after which the shell checks for changes in the modification time (**mtime**) of any of the files specified in the **$MAILPATH** or **$MAIL** variables. The default value is 600 seconds (ksh) or 60 seconds (bash). We use the ksh value. When the time has elapsed the shell checks for mail before outputting the next primary prompt.

MAILPATH
A colon-separated list of file names to check after **$MAILCHECK** seconds has elapsed. Each file name can be followed by a '?' and a message that is printed if the file is modified. The message undergoes parameter expansion, command substitution, and arithmetic expansion and the **$_** variable contains the name of the file that has changed. The default message is **"you have mail in $_"** (ksh).

OLDPWD
The previous working directory set by the **cd** builtin.

OPTARG
The value of the last option argument processed by the **getopts** builtin utility.

OPTIND
The index of the last option argument processed by the **getopts** builtin utility.

OPTERR
If set to 1, display errors generated by the **getopts** builtin. This variable is a non-POSIX bash extension.

OSTYPE
The type of operating system the shell is running on, determined at compilation time. This variable is a non-POSIX extension.

PATH The search path for commands. **$PATH** cannot be changed if the shell is in the restricted mode. The default value is **"/bin:/usr/bin:/sbin:/usr/sbin"**.

PPID The **PID** of the parent of this shell.

PROMPT_COMMAND
If set, the value is a command to be executed before printing the primary prompt **$PS1**. This variable is a non-POSIX bash extension.

PROMPTCHARS
If set to a two-character string, this variable is used when printing prompt strings. The first character is used for normal users, the second character is used for root. The default value is **"$#"**. This is a non-POSIX extension that behaves like tcsh's **promptchars** variable.

PS0 The value of this variable is expanded for parameter expansion, command substitution, and arithmetic expansion and the result printed on stderr, right before each simple command is executed. This variable is a non-POSIX bash extension.

PS1 The primary prompt string. The value of this variable is expanded for parameter expansion, command substitution, and arithmetic expansion. The character '!' in **$PS1** is replaced by the history number of the current command. Two bangs '!!' produce a single '!' when the prompt string is printed. The default value is **"[\u \W]\$\ "**.

PS2 The secondary prompt string, defaults to **"> "**.

PS3 The third (selection) prompt string used within a select loop, defaults to **"#? "**.

PS4 The value of this variable is expanded for parameter evaluation, command substitution, and arithmetic expansion and precedes each line of an execution trace. The default value is **"+ "**.

PWD The present (current) working directory set by the **cd** builtin.

RANDOM
Each time this variable is referenced, a random integer number, uniformly distributed between 0 and 32767, is returned. The sequence of random numbers can be initialized by assigning a numeric value to **$RANDOM**. This variable is a non-POSIX extension.

REPLY
This variable is set by the **select** statement and by the **read** builtin utility when no arguments are passed to it. This variable is a non-POSIX extension.

SAVEDIRS
If set, this variable means a login shell must save the directory stack in the file indicated by the value of **$DIRSFILE**. This variable is similar to tcsh's **$savedirs** variable. This variable is a non-POSIX extension.

SECONDS
Each time this variable is referenced, the number of seconds since the shell started is returned. If this variable is assigned a value, the value returned is the value that was assigned plus the number of seconds since the assignment. This variable is a non-POSIX extension.

SHELL
The full pathname of the shell. Upon startup, if the basename of this variable is rsh, rlsh, or lrsh, the shell becomes restricted. **$SHELL** is not typically set by the shell.

SHELLOPTS
A colon-separated list of enabled shell options, the ones that are reported as *on* when invoking **set −o**. If this variable is set on shell startup, each option is enabled in the shell. This variable is a readonly non-POSIX bash extension.

SHLVL
This variable is assigned the value of 1 the first time the shell is run. Its value gets incremented by 1, each time a new instance of the shell is started. This variable is a readonly non-POSIX bash extension (although bash doesn't mark it as readonly).

SUBSHELL
This variable is incremented by 1 in each subshell the shell invokes. This variable is a non-POSIX extension.

TIMEFORMAT
The format string specifying how the timing information for pipelines prefixed with the **time** command should be displayed. The possible format sequences are:

%%	A literal %.
%[p][l]R	The elapsed (real) time in seconds.
%[p][l]U	The number of CPU seconds spent in user mode.
%[p][l]S	The number of CPU seconds spent in system mode.
%P	The CPU usage percentage, computed as (U + S) / R.

In all the above format sequences, braces denote optional portions. The optional *p* is a digit specifying the precision, i.e. the number of fractional digits after a decimal point. A value of 0 means no decimal point or fraction is to be output. A maximum of three places after the decimal point can be displayed. If *p* is greater than 3 or is not specified, it is treated as 3. The optional *l* (ell) specifies a longer format which includes hours, minutes, and seconds of the form **HHhMMmSS.FFs**. The value of **l** determines whether or not the fraction **.FF** is included. All other characters are output without change, and a trailing newline is added afterwards. If **$TIMEFORMAT** is unset, the default value is "*\nreal\t%2lR\nuser\t%2lU\nsys\t%2lS*" (ksh and bash, except that bash uses a precision of 3 instead of 2). If the value of **$TIMEFORMAT** is null, no timing information is displayed.

TMPDIR
The pathname of the directory used to create temporary files. If not set, the default is **/tmp**. This variable is a non-POSIX bash extension.

TMOUT
The default time-out value for the **read** and **select** builtin utilities. The shell terminates if a line is not entered within the prescribed number of seconds while reading from a terminal. This variable is a non-POSIX extension used by ksh and bash.

UID The numeric user id of the currently logged user. This variable is a non-POSIX extension.

USER The username of the currently logged user. This variable is a non-POSIX extension.

VISUAL
If the value of this variable matches the pattern *[Vv][Ii]*, the **vi** option is turned on. Currently, no other patterns are recognized by the Layla shell. The value of **$VISUAL** overrides the value of **$EDITOR**.

THE SHELL PROMPT

When running interactively, the shell prompts the user by prinintg the value of **$PS1** to stderr after expanding it for parameter expansion, command substitution, and arithmetic expansion, before reading the next command. Each command is not considered complete, the secondary prompt **$PS2** is expanded and printed to stderr.

CONDITIONAL EXPRESSIONS

Conditional expressions are used with the '[[' compound command to test files' attributes and compare strings. Field splitting and pathname expansion are not performed here.

Conditional expressions can be formed by one or more of the following unary and binary expressions:

−a *f* True if file *f* exists. This option is the same as **−e**.

−b *f* True if file *f* exists and is a block special file.

−c *f* True if file *f* exists and is a character special file.

−d *f* True if file *f* exists and is a directory.

−e *f* True if file *f* exists.

−f *f* True if file *f* exists and is a regular file.

−g *f* True if file *f* exists and it has its **setgid** bit set.

−G *f* True if file *f* exists and its group id matches the effective group id of this process.

−h *f* True if file *f* exists and is a symbolic link.

−k *f* True if file *f* exists and it has its sticky bit set.

−L *f* True if file *f* exists and is a symbolic link.

−n *str* True if the length of *str* is non-zero.

−N *f* True if file *f* exists and its modification time is greater than its last access time.

−o *op* True if option named *op* is on.

−o *?op* True if *op* is a valid option name.

−O *f* True if file *f* exists and is owned by the effective user id of this process.

−p *f* True if file *f* exists and is a FIFO special file or a pipe.

−r *f* True if file *f* exists and is readable by current process.

−s *f* True if file *f* exists and has size greater than zero.

−S *f* True if file *f* exists and is a socket.

−t *fd* True if file descriptor *fd* is open and associated with a terminal device.

−u *f* True if file *f* exists and it has its **setuid** bit set.

−w *f* True if file *f* exists and is writable by current process.

−x *f* True if file *f* exists and is executable by current process.

−z *str* True if the length of *str* is zero.

file1 **−ef** *file2*
True if both *file1* and *file2* exist and refer to the same file.

file1 **−nt** *file2*
True if *file1* exists and *file2* doesn't, or if *file1* is newer than *file2*.

file1 **−ot** *file2*
True if *file2* exists and *file1* doesn't, or if *file1* is older than *file2*.

string True if *string* is not null.

string **==** *pattern*
True if *string* matches *pattern*.

string **=** *pattern*
Same as '=='.

string **!=** *pattern*
True if *string* does not match *pattern*.

string **=~** *pattern*
True if *string* matches *pattern*.

string1 **<** *string2*
True if *string1* comes before *string2* based on the ASCII value of the strings' characters.

string1 **>** *string2*
True if *string1* comes after *string2* based on the ASCII value of the strings' characters.

expr1 **–eq** *expr2*
True if *expr1* is equal to *expr2*.

expr1 **–ge** *expr2*
True if *expr1* is greater than or equal to *expr2*.

expr1 **–gt** *expr2*
True if *expr1* is greater than *expr2*.

expr1 **–le** *expr2*
True if *expr1* is less than or equal to *expr2*.

expr1 **–lt** *expr2*
True if *expr1* is less than *expr2*.

expr1 **–ne** *expr2*
True if *expr1* is not equal to *expr2*.

Compound expressions can be constructed from these primitives by using any of the following operators (the brace operator is currently not supported in this shell):

(expression)
True if *expression* is true. Used to group expressions.

!expression
True if *expression* is false.

expression1 && **expression2**
True if *expression1* and *expression2* are both true.

expression1 || expression2
True if either *expression1* or *expression2* is true.

INPUT AND OUTPUT

Command input and output can be redirected using special shell constructs, which can precede or follow the command name, and which are not passed to the invoked command. Command substitution, parameter expansion, and arithmetic expansion occur and the result is used for redirecting input and output streams as appropriate. Field splitting is not performed on the results of expansion. If the redirected file is of the form */dev/tcp/host/port* or */dev/udp/host/port*, the shell attempts to make a tcp or udp connection to the corresponding host and port number. No intervening space is allowed between the redirection operator and its operands.

<word Redirect stdin to read from the file indicated by **word**.

>word Redirect stdout to read from the file indicated by **word**. If the file does not exist then it is created. If the file exists and the **noclobber '–C'** option is set, an error results. Otherwise, the file is truncated to zero length.

>|word Similar to **>**, except that it overrides the **noclobber** '**–C**' option.

>>word
Redirect output to the file indicated by *word*. If the file exists, append output to the end-of-file. Otherwise, create the file.

<>word
Open the file indicated by *word* for read/write on stdin.

<<[-]word
Redirect stdin to read from a here-document. The shell reads up to the line that contains *word* after any quoting has been removed. No parameter substitution, command substitution, arithmetic expansion or pathname expansion is performed on *word*. The resulting document (the here-document), becomes the input. If any character of *word* is quoted, no expansion is done on the document. Otherwise, parameter expansion, command substitution, and arithmetic expansion occur, the **\newline** sequence is ignored, and '\' must be used to quote the characters '\', '$', and '`'. If '-' is appended to the '<<' operator, all leading tabs are stripped from the document.

<<<word
Redirect stdin to read from a here-string. *word* becomes the contents of the here-document after parameter expansion, command substitution, and arithmetic expansion is performed.

<&digit
Duplicate stdin to read from file descriptor *digit*.

>&digit
Duplicate stdout to write to file descriptor *digit*.

<&digit-
Move file descriptor *digit* to stdin (i.e. duplicate and close the file descriptor).

>&digit-
Move file descriptor *digit* to stdout (i.e. duplicate and close the file descriptor).

<&- Close stdin.

>&- Close stdout.

<&p Redirect stdin to read from the co-process start by the **coproc** builtin.

>&p Redirect stdout to come from the co-process start by the **coproc** builtin.

<#((expr))
Evaluate arithmetic expression *expr* and move file descriptor 0's (stdin) offset to the resulting bytes from the start of the file.

>#((expr))
The same as *<#* for file descriptor 1 (stdout).

If a redirection operator is preceded by a digit, the file descriptor to be redirected is that referred to by the digit (instead of the default 0 or 1). If a redirection operator is preceded by {**varname**} with no intervening spaces, a file descriptor greater than 10 is selected by the shell and stored in the variable named **varname**.

THE ENVIRONMENT

The environment is a list of name-value string pairs that is passed down to commands when they are being executed. On startup, the shell scans the environment and creates a shell variable for each entry, giving it the corresponding value and marking it for export. Commands executed by the shell inherit the environment list. If the user modifies the values of these variables or creates new ones by using the **export** builtin utility, these new variables become part of the environment. Variable assignments occurring before the command name are added to the command's environment, while those occuring after the command name are only considered if the **keyword** '**−k**' option is set.

JOBS

If the **monitor** '**−m**' option is set, an interactive shell associates a job with each pipeline. The shell keeps a table of current jobs, each one being assigned a small integer number. When a job is started asynchronously (in the background) by affixing it with '&', the shell prints a line that looks like:

[1] 12345

which indicates that a job was started asynchronously with job number 1, and that the last process in the pipeline had a **PID** of 12345.

To suspend a running job, press **CTRL-Z** to send the **STOP** signal to the current job. The shell displays a message informing your of the change in the job's status before printing the primary prompt. The job can then be run in the background or foreground by invoking the **bg** and **fg** builtins, respectively.

If a background job tries to read from the terminal, it is sent the **SIGTTIN** signal, which suspends the job by default. Background jobs can usually send output to the terminal, unless the **stty** was used to modify this behavior, in which case the job is sent a **SIGTTOU** signal when it tries to write to the terminal.

Jobs are referred to by using the **PID** of any process in the job, or by the job id which can take the following formats:

%number
The job with the given number.

%string
Any job whose command line begins with *string*.

%?string
Any job whose command line contains *string*.

%% The current job.

%+ The current job. Equivalent to %%.

%- The previous job.

When a process changes state, the shell is sent a **CHLD** signal. The shell outputs the list of stopped jobs right before outputting the next primary string. If the **notify** '**−b**' option is set, the shell notifies the user of any change in job's status immediately. When a background job finishes and the **monitor** '**−m**' option is set, the shell executes any set **CHLD** traps.

When the user tries to exit an interactive shell while there are running or stopped jobs, the shell prints a warning message. If another attempt is made at exit, the jobs are sent a HUP signal before the shell exits. When a login shell receives a HUP signal, it re-sends the HUP signal to all job that have not been disowned with the **disown** builtin utility.

SIGNALS

The interactive shell ignores the **TERM** and **QUIT** signals. If the **monitor** '**−m**' option is set, the **TSTP**, **TTIN**, and **TTOU** signals are also ignored. The **CHILD**, **INT**, **WINCH** and **HUP** signals are caught and handled.

For background (asynchronous) pipelines, the **INT** and **QUIT** signals are ignored if the **monitor** '**−m**' option is not active. In all other cases, signals have the values inherited by the shell from its parent upon startup.

COMMAND EXECUTION

Commands that doesn't contain slashes are looked up. If the command name matches the name of a special builtin utility, the utility is executed in the current shell. If not, the command name is checked against user-defined functions. If it matches a defined function, positional parameters are saved and then reset, before assigning them to the arguments passed as part of the function call. Functions are also executed in the current shell. Positional parameters are restored when the function returns. The exit status of a function call is the value of the last command executed in the function. If a command name is neither a special builtin utility nor a user-defined function, the shell looks for a regular builtin utility with the given name. If found, the regular builtin utility is executed in the current shell.

If all the above fails, the shell looks for commands external to the shell's executable. The shell variable **$PATH** defines the search path which the shell uses to look for a directory containing the command. It consists of a colon-separated list of directory names. The default path defined by Layla shell is **"/usr/local/bin:/usr/local/sbin:/usr/bin:/usr/sbin"**. Each directory in the path is searched for an executable file with the given name. If found, a subshell is forked and the command is executed by calling **exec**().

One of the distinctive features of lsh's commandline interpreter is that it performs two passes on the input source (whether it was an interactively-entered command, a script file, or a command string). The first pass is in the frontend (the parser), which tokenizes input and forms an abstract source tree (AST) out of it. The second pass is in the backend (the executor), which executes the AST. That two-tiered (or two pass) layout simplifies the process of reading and executing commands within the shell, but it also has some side effects. One of these side effects is that syntax errors in a shell script will result in the shell refraining from executing the whole script. Another side effect is that functions can be defined anywhere in the script, and they can be called before they are defined (provided the definition occurs in the same script file). So for example, the following script will raise an error in bash, but not in lsh:

```
# the following line will execute successfully in lsh, but not in bash
```

```
f1

# function definition can follow function call in the same script file
f1 ()
{
  echo "Inside f1"
}
```

Another side effect is that empty function bodies are permissible in lsh, while they are not so in bash. For example, in the following script, @code{echo} will be executed in lsh, while it a syntax error will result in bash:

```
empty ()
{
}

echo "Hi there!"
```

A third side effect is that function nesting is permissible in lsh, which is also the case in bash. The difference here is that because lsh parses the whole script before passing it to the executor, a nested function is recognized and added to the function table at the time its parent function is parsed. This doesn't mean that either of the two functions is being processed or its commands executed; just the function names are added to the global function table. Again, this is not the case in bash.

For example, in the following script, both f1 and f2 are added to the function table by the time the parser finishes with the script. This is why the three function calls will work properly in lsh, while in bash the first call to f2 will result in a "command not found..." error; the second call to f2 and the call to f1 will both work, as the call to f1 resulted in the addition of f2 to the function table:

```
f1 ()
{
   f2 ()
   {
      echo "function f1"
   }
}

f2    # results in "command not found..." error in bash, but not lsh
f1    # works in both bash and lsh
f2    # works in both bash and lsh
```

COMMAND HISTORY

The last **$HISTSIZE** (which defaults to 512) commands entered in an interactive shell is saved in the history file, whose path is given by **$HISTFILE**. If **$HISTFILE** is unset of null, or the file is not writeable by the shell, **$HOME/.lsh_history** is used instead.

The builtin utility **history** can be used to list the history entries, while the builtin **fc** can be used to edit the list. When calling **fc** to edit history commands, the **−e** option can be used to specify the editor program to pass the history list to. If no editor is specified, **fc** consults the **$FCEDIT** variable and invokes the command named in the value of this variable. If **$FCEDIT** is not set, **$HISTEDIT** is consulted. If this is also not set, **$EDITOR** is consulted. If all of this fails, the default value **/bin/ed** is used. The edited commands are printed and executed upon leaving the editor. The **−s** option causes **fc** to skip the editing phase and to re-execute the commands directly.

VI EDITING MODE

The Layla shell provides builtin **vi** editing functionality. There are two default modes. When the shell starts, the user is placed in the *input mode*. To edit commands, the user must press **ESC** (ASCII 033) to enter the *control mode*. Most control commands accept an optional repeat count that is entered before the command character.

The Input Mode

The default editor mode is the input mode. The following edit commands are supported in this mode:

ERASE
The default is **^H** or **Backspace**. Deletes the previous character.

^W Deletes the previous blank-separated word.

EOF If the first character in the line, it causes the shell to terminate, unless the **ignoreeof** option is set, or the **$IGNOREEOF** variable is set to a numeric value greater than zero.

^V Removes the next character's editing features, if any.

**** Escape the next **ERASE** or **KILL** character.

^I, **TAB**
Attempts command, variable, username, hostname, or filename completion.

The Control Mode

When the user presses **ESC** and the **vi** option is set, the shell's editor enters the control mode. The following commands are recognized in this mode.

Motion Edit Commands

[count]**l**
Move the cursor one character forward (to the right).

[count]**[C**
Move the cursor one character forward (to the right).

[count]**w**
Move the cursor one word forward.

[count]**W**
Move the cursor to the beginning of the next word following a blank.

[count]**e**
Move the cursor to the end of this word.

[count]**E**
Move the cursor to the end of the current blank-delimited word.

[count]**h**
Move the cursor one character backwards (to the left).

[count]**[D**
Move the cursor one character backwards (to the left).

[count]**b**
Move the cursor one word backwards.

[count]**B**
Move the cursor to the preceding blank-separated word.

[count]**|**
Move the cursor to column number **count**.

[count]**fc**
Find the next character **c** in the current line.

[count]**Fc**
Find the previous character **c** in the current line.

[count]**tC**
Equivalent to **f** followed by **h**.

[count]**Tc**
Equivalent to **F** followed by **l**.

[count]**;**
Repeat **count** times the last find command (**f, F, t, or T**).

[count]**,**
Reverse the last find command **count** times.

0 Move the cursor to the start of line.

^ Move the cursor to start of line.

[H Move the cursor to the first non-blank character in line.

$ Move the cursor to the end of line.

[Y Move the cursor to the end of line.

% Moves to balancing brace or bracket.

Search Edit Commands

[count]**k**
Fetch a previous command, **count** entries back.

[count]**-**
Fetch a previous command, **count** entries back. Equivalent to **k**.

[count]**[A**
Fetch a previous command, **count** entries back. Equivalent to **k**.

[count]**j**
Fetch the next command, **count** entries forward.

[count]**+**
Fetch the next command, **count** entries forward. Equivalent to **j**.

[count]**[B**
Fetch the next command, **count** entries forward. Equivalent to **j**.

[count]**G**
Fetch command number **count**. The default is the last history entry.

/string Search history backwards for a command containing **string**. If **string** start with '^', the matched line must begin with **string**. If **string** is null, the last entered **string** is used.

?string Search history forwards for a command containing **string**. If **string** starts with '^', the matched line must begin with **string**. If **string** is null, the last entered **string** is used.

n Search backwards for the next match of the last pattern to '/' or '?'.

N Search forwards for the next match of the last pattern to '/' or '?'.

Text Modification Edit Commands

a Enter input mode and enter text after the current character.

A Append text to the end of line.

[count]**c**motion

c[count]**motion**
Delete all characters from the current character up to the character that **motion** would move the cursor to, and then enter the input mode. If **motion** is **c**, the entire line is deleted and the input mode is entered.

C Delete all characters from the current character up to the end of line and then enter the input mode.

S Equivalent to **cc**.

[count]**s**
Replace characters under the cursor in the input mode.

D[count]**d**motion
Delete all characters from the current character up to the end of line.

d[count]**motion**
Delete all characters from the current character up to the character that **motion** would move the cursor to. If **motion** is **d**, the entire line is deleted.

i Enter the input mode and insert text before the current character.

I Insert text before the beginning of the line.

[count]**P**
Yank (paste) text before the cursor.

[count]**p**
Yank (paste) text after the cursor.

R Enter the input mode and replace characters on the screen with characters to be typed, as if the user pressed the **INSERT** key.

[count]**r**c
Replace **count** characters, starting at the cursor position, with **c**, then advance the cursor.

[count]**x**
Delete the current character.

[count]**X**
Delete the preceding character.

[count]**.**
Repeat the previous text modification command.

[count]**~**
Invert the case of **count** characters, starting at the cursor position, then advance the cursor.

[count]**_**
Append the **count** word of the previous command to the line and enter the input mode. The last word is used if **count** is omitted.

***** Append a '*' to the current word and attemp file name completion. If no match is found, the bell is ringed. Otherwise, the word is replaced by the match and input mode is entered.

**** Perform command or file name completion.

Other Edit Commands

[count]**y**motion

y[count]**motion**
Yank (copy) all characters from the current character up to the character to which **motion** would move the cursor. Yanked characters are saved in a special buffer. The cursor position is not changed.

yy Yank (copy) the current line.

Y Yank (copy) the current line from the cursor position to the end of the line.

U Undo all the text modifying commands performed on current line.

[count]**V**
Return the command:

fc -e ${VISUAL:-${EDITOR:-vi}} count

in the input buffer. If **count** is omitted, the current line is used.

^L Print a line feed and then print the current line. Works in control mode.

^J Print a new line and then execute the current line. Works in both modes.

^M Return and execute the current line. Works in both modes.

Comment or uncomment the current line.

@letter
Search for an alias by the name **letter**. If an alias of this name is found, insert its value on the input buffer.

^V Display the version of the shell.

BUILTIN UTILITIES

The following simple commands are executed in the shell process, i.e. no subshell is invoked. Input and output redirection is allowed and affects the current shell. Except for **:**, **true**, **false**, **echo**, **newgrp**, and **login**, all builtin utilities accept '--' to indicate the end of options and start of arguments. Unless otherwise stated, the **−h** option prints a short history and usage message, while the **−v** option prints the utility (i.e. the shell)'s version.

: [*arg* ...]
Expand parameters only.

. name *[arg* ...]
If **name** is a defined function, the function is executed in the current shell environment. If **name** refers to a file, the file is read and the commands are executed in the current shell environment. If any arguments are specified, they become the positional parameters for the command. The exit status is the exit status of the last command executed.

alias [−hv] [*name*[=*value*]] ...
With no arguments, prints the list of aliases in the form of **'name=value'** on stdout. When one or more arguments are specified, an alias is defined for each **name** whose **value** is specified. For each name in the argument list for which no value is specified, the name and value of the alias is printed. The exit status is non-zero if a **name** is specified for which no alias is defined.

bg [−hv] [*job* ...]
If job control is on (the **monitor** option is set), put each specified *job* in the background. If no *job* is specified, the current job is put in the background. See the *Jobs* section for a description of the format of *job*.

break [*n*]
Exit from the *n*-th enclosing **for**, **while**, **until**, or **select** loop, if any. If *n* is not specified, it defaults to 1.

bugreport
Send bugreports to the shell's author(s).

builtin [**-hvsra**] [*name* [*args* ...]]
If **name** is not specified, a list of builtins is printed on stdout. The **−s** option prints only the special builtins. **name** is the name of a shell builtin utility to invoke. **args** are the arguments to pass to the builtin utility. The **−s** option prints the special builtins, The **−r** option prints the regular builtins, and the **−a** option prints both.

caller [*n*]
Print the context of any active subroutine call. **−n** is a non-negative integer denoting one of the callframe in the current call stack. The current frame is 0. Each call to a function or dot script results in a new entry added to the call stack.

cd [**−h**] [**−nplv**] [**−L|-P**] [*directory*]

cd [**−h**] [**−nplv**] [**−L|-P**] [-]

In the first form **cd** changes the current directory to the given *arg*. If *arg* is a hyphen '−', the directory is changed to the previous directory. If no *arg* is given, shell variable **$HOME** is used. Shell variable **$PWD** is set to the current directory. Shell variable **$CDPATH**, if set, gives the search path for the directory containing **arg**. It is composed of a colon-separated list of directory names. If **arg** begins with a '/', **$CDPATH** is not used.

Symbolic link names are treated literally when parsing directory names. This is what the **−P** option does (physical treatment of links). The **−L** option causes symbolic links to be resolved when determining the directory (logical treatment of links). If both options are supplied, the last instance of on the command line determines which option is used. Restricted shells cannot execute **cd**.

The **−l**, **−n**, **−p**, and **−v** options have the same meaning as when used with the **dirs** builtin. They all imply **−p**.

command [**−hp**] *name* [*arg ...*]

command [**−hp**] [**-v|-V**] *name*

Without passing the **−v** or **−V** options, **command** executes **name** with *arg* as arguments. The **−p** option forces the use of a default path guaranteed to find all commands on the system. The **−v** option prints a description of **name** if used as a command name within the shell. The **−V** option prints a more verbose output.

continue [*n*]

Resume the next iteration of the *n*-th enclosing **for**, **while**, **until**, or **select** loop. If *n* is not specified, it defaults to 1.

coproc command [*redirections*]

Execute *command* in a coprocess (subshell with pipe). *redirections* are optional file redirections. A pipe is opened between the shell and the coprocess before any redirections are performed. Shell variable **$COPROC_PID** contains the *PID* of the coprocess. Shell variable **$COPROC0** points to the reading end of the pipe (connected to command's stdout), while variable **$COPROC1** points to the writing end of the pipe (connected to command's stdin). You can feed output to the process by invoking:

$ cmd >&p

Similarly, you can read the process's output by invoking:

$ cmd <&p

declare [**−hvfFgrxlut**] [**−p**] [*name*=[*value*]...]

Declare variables and give them attributes. *name* is the name of the variable to which an attribute or value is set, and *value* is the value to give to the variable. The **−f** option restricts output to shell functions. The **−F** option doesn't print function definitions. The **−g** option declares/modifies variables at the global scope. The **−l** option converts all characters in variable's value to lowercase on assignment. The **−p** option prints the attributes and values of each *name*. The **−r** option marks each *name* as readonly. The **−t** option gives functions the **trace** attribute (doesn't work on variables). The **−u** option converts all characters in variable's value to uppercase on assignment. The **−x** option marks each *name* for export.

dirs [**−hclpv**] [*+N* | *-N*]

dirs -S|-L [*filename*]

Display the contents of the directory stack. The **+N** argument prints the N-th directory from the top (the left side of the printed list), counting from zero (which is the current working directory). The **−N** argument prints the N-th directory from the bottom (the right side of the printed list), counting from zero (which is the first dir pushed on the stack). *filename* is the file to save/load the directory stack to/from. The **−c** option clears the stack, i.e. removes all directories. The **−l** option prints full pathnames, it doesn't use ~ to indicate the home directory. The **−L** option loads the directory stack from the given filename. If no filename is supplied, uses **$DIRSFILE** or defaults to

~/.lshdirs. The **−n** option wraps entries before they reach edge of the screen. The **−p** option prints each directory on a separate line. The **−S** option saves the directory stack to the given filename. If no filename is supplied, uses **$DIRSFILE** or defaults to *~/.lshdirs*. The **−v** option prints each directory with its index on a separate line.

disown [**−arsv**] [**−h**] [*job ...*]
Do not to send a HUP signal to each specified *job*, or to all active jobs if *job* is omitted, when a login shell exits. The **−a** option disown all jobs. The **−h** option doesn't remove jobs from the jobs table, only marks them as disowned. The **−r** option disowns only running jobs, while **−s** option disowns only stopped jobs.

dump [**-hv**] [*argument ...*]
Dumps memory values of the passed arguments. **argument** can be one of the following:

symtab will print the contents of the local symbol table
vars will print out the shell variable list (similar to `declare -p`)

echo [**−enE**] [*arg ...*]
Print each of the arguments separated by a space and terminated by a newline. The **−e** option allows escaped characters in **arg**s. The **−E** option doesn't allow escaped characters. The **−n** option suppresses newline echoing.

enable [**−ahnprsv**] [*name ...*]
Enables/disables shell builtins. *name* is the name of a shell builtin utility to enable or disable. The **−a** option prints a list of all builtins, enabled and disabled. The **−n** option disables each listed builtin. The **−p** option prints a list of enabled builtins. The **−r** option prints a list of enabled and disabled regular builtins. The **−s** option print a list of enabled and disabled special builtins.

eval [*arg ...*]
The arguments are read as input to the shell and the resulting commands are executed.

exec [**−cl**] [**−a** *arg0 ...*] [*arg ...*]
If one or more *arg* is specified, the command is executed in place of this shell without forking a new process. The **−c** option clears the environment before applying variable assignments associated with the **exec** invocation. The **−a** option causes **arg0**, rather than **arg[0]**, to become the first argument for the new process. If *arg* is not specified, **exec** modifies the file descriptors of the shell, as indicated by the input/output redirection list. The **−l** option places a dash in front of **argv[0]**, just as the **login** utility does.

exit [*n*] Exit the shell with the exit status *n*. The value is the least significant 8 bits of *n*. If *n* is omitted, the exit status is that of the last command executed.

export [**−hvn] [−p**] [*name*[=*value*]] ...
If *name* is not specified, the names and values of each variable with the export attribute are printed with the values quoted to allow reentry to the shell. The **−p** option causes the word **export** to be printed before each name. Otherwise, the specified names are marked for export to the environment of commands executed by the shell. The **−n** option removes the export attribute of the passed variable **name**s.

false Return false result (non-zero exit status).

fc [**−hvr**] [**−e** *editor*] [*first* [*last*]]

fc −l [−hvnr] [*first* [*last*]]

fc −s [−hv] [*old*=*new*] [*first*]
In the first form, a range of commands from *first* to *last* is selected from the history list. *first* and *last* can be specified as numbers or strings. A string finds the most recent command starting with the given string. A negative number gives an offset, relative to the current command number. In the third form, *old*=*new* replaces the first occurence of *old* with *new* in the command. The **−e** option specifies the editor to use when editing commands, which is passed as the *editor* argument. *editor* is invoked on a temporary file containing the selected commands. If *editor* is not supplied, the

value of **$FCEDIT** is used. If this is null or empty, the value of **$HISTEDIT** is used. If null or empty, the value of **$EDITOR** is used. Otherwise, the default editor (*/bin/ed*) is used. When editing finishes, the edited commands are executed. If *last* is not specified, it is set to *first*. If *first* is not specified, it defaults to the previous command for editing and -16 for listing. The **−l** option lists commands instead of invoking them. The **−n** option suppresses command numbers when listing. The **−r** option reverses the order of listed/edited commands. The **−s** option re-executes commands without invoking the editor.

fg [**−hv**] [*job ...*]
If job control is on (the **monitor** option is set), put each specified *job* in the foreground. If no *job* is specified, the current job is put in the foreground. See the *Jobs* section for a description of the format of *job*.

getopts optstring varname [*arg ...*]
Check *arg* for legal options. If *arg* is omitted, positional parameters are used. An option argument can begin with a '+' or '-'. An option that does not begin with '+' or '-', or the special argument '--', ends the options list. Options beginning with '+' are only recognized when the **optstring** begins with a '+'. The **optstring** contains letters that **getopts** recognizes as valid options. If a letter is followed by ':', that option is expected to have an argument. Options can be separated from their arguments by spaces. **getopts** places the next option letter in the variable **varname**. The option letter is preceded by a '+' when *arg* begins with a '+'. The index of the next *arg* is stored in the shell variable **$OPTIND**. The option argument, if any is found, is stored in **$OPTARG**. A leading ':' in **optstring** causes **getopts** to store the letter of an invalid option in **$OPTARG** and set the value of **varname** to '?' for an unknown option, and to ':' when a required option argument is missing. Otherwise, getopts prints an error message. The exit status is non-zero when options are finished. Options ':', '+', '-', '?', '[', and ']' are not allowed.

glob [**−eE**] [*args ...*]
Echoes *args*, delimited by NULL characters. The **−e** option allows escaped characters in arguments. The **−E** option doesn't allow escaped characters in arguments. This utility doesn't recognize the **−n** option as does **echo**.

hash [**−hvld**] [**−p** *path*] [**−r**] *utility ...*

hash −a
Remembers/reports utility locations. *utility* is the name of a utility to search and add to the hashtable. The **−a** option forgets, then re-searches and re-hashes all utilities whose names are currently in the hashtable. The **−d** option forgets the location of each passed *utility*. The **−l** option prints the list of hashed utilities and their paths. The **−p** option performs utility search using *path* instead of the **$PATH** variable. The **−r** option forgets all previously remembered utility locations.

help [**−ds**] [*command*]
Shows help for builtin utilities and commands. The **−d** option prints a short description of each *command*. The **−s** option prints the usage or synopsis of each *command*.

history [**−hR**] [*n*]

history −c

history −d *offset*

history −d *start−end*

history [**−anrwSL**] [*filename*]

history −ps *arg ...*
Prints command history. The *n* argument prints only the last *n* lines. The **−a** option appends the in-memory history list to *filename*. If *filename* is not supplied, the default history file is used. The **−c** option clears the history list. The **−d** option deletes the history entry at position *offset*. Negative offsets count from the end of the list; offset -1 is the last command entered. The **−d** option deletes history entries between offsets *start* and *end*, which can be negative, as described above. The **−h** option prints history entries without leading numbers. The **−L** option is equivalent to **−r**. The **−n**

option appends the entries from *filename* to the in-memory list. If *filename* is not supplied, the default history file is used. The **−p** option performs history substitution on *arg*s and prints the result on stdout. The **−r** option reads the history file and append the entries to the in-memory list. The **−R** option reverses the listing order (most recent entries are printed first). The **−s** option adds *arg*s to the end of the history list as one entry. The **−S** option is equivalent to **−w**. The **−w** option writes out the current in-memory list to *filename*. If *filename* is not supplied, the default history file is used.

hup [*command*]
Runs a command so that it can receive **SIGHUP** signal. Commands can override this by defining their own signal handlers.

jobs [**−hnrsv** [**−l** | **−p**] [*job ...*]

jobs -x command [*argument ...*]
In the first form, lists information about each *job*, or all active jobs if *job* is omitted. The **−l** option lists jobs' PIDs in addition to the normal information. The **−n** option only displays jobs that have stopped or exited since the last notification. The **−p** option lists only the process groups ids. The **−r** option reports only running jobs, while the **−s** option reports stopped jobs. See the *Jobs* section for a description of the format of *job*.

In the second form (with **−x**), replaces all occurences of *job*s in **command** and **argument**s with the process group ID of the respective job, then runs **command**, passing it the given **arguments**.

kill [**−s signame**] *job ...*

kill [**−n signum**] *job ...*

kill −l | **−L** [*sig ...*]

kill [**−signame**] *job ...*

kill [**−signum**] *job ...*
Send either the **TERM** signal or the specified signal to the given jobs or processes. Signals are either specified as numeric arguments to the **−n** option, or by symbolic names to the **−s** option, without the SIG prefix. The **-n** and **-s** options can be omitted and the signal number or name placed immediately after the '-'. See the *Jobs* section for a description of the format of **job**. In the third form, if *sig* is not specified, signal names are listed. Otherwise, for each symbolic *sig*, the corresponding signal number is listed. For each numeric *sig*, the corresponding signal name is listed.

let [*arg ...*]
Evaluate each *arg* as an arithmetic expression. See the *Arithmetic Evaluation* section for a description of arithmetic expression evaluation. The exit status is 0 if the value of the last expression is non-zero, 1 otherwise.

local *name*[=*word*] ...
Define local variable *name*, setting the local attribute to it and giving it the value *word*.

logout [*n*]
Exit a login shell, returning *n* as the exit status code.

mailcheck [**−hvq**]
Check for mail at specified intervals. The **−q** causes **mailcheck** not to output messages in case of error or no mail available.

memusage arg...
Show the shell's memory usage. Each **arg** shows the memory allocated for a different shell internal structure, which can be one of the following:

aliases	memory allocated for alias names and values
cmdbuf, cmdbuffer	memory allocated for the command line buffer
dirstack	memory allocated for the directory stack

hash, hashtab	memory allocated for the commands hashtable
history	memory allocated for the command line history table
input	memory allocated for the currently executing translation unit
stack, symtabs	memory allocated for the symbol table stack
strbuf, strtab	memory allocated for the internal strings buffer
traps	memory allocated for the signal traps
vm	memory usage of different segments (RSS, stack, data)

The **–l** option shows long output (i.e. prints more details).

newgrp [**–hv**] [**–l**] [*group*]
Create a new group and restart the shell in a new execution environment. *group* is the group name (or ID) to which the real and effective group IDs shall be set. The **–l** option changes the environment to a login environment.

nice [**+n**] [*command*]

nice [**–n**] [*command*]
Run a command with the given priority. *n* can be positive or negative, specifying the nice priority to give to *command*, or the shell if no *command* is given (the plus sign can be omitted for positive nice values). On many systems, only **root** can pass negative nice values. *command* is the command to run under priority *n*, which must be an external command.

nohup [*command*]
Run a command, ignoring **SIGHUP**. As with **nice**, **command** must be an external command.

notify [*job ...*]
Notify immediately when jobs change status. **job** is the job id of the job to mark for immediate notification. See the *Jobs* section for a description of the format of **job**.

popd [**–chlnpsv**] [*+N | −N*]
Pop directories off the stack and **cd** to them. If *N* is positive, it removes the N-th directory, counting from 0 from the left. If it is negative, it removes the N-th directory, counting from 0 from the right. If called without arguments, **popd** removes the top directory from the stack and calls **cd** to change the current working directory to the new top directory (equivalent to `popd +0`. The **–c** option manipulates the stack without **cd**ing to the directory. The **–s** option suppresses the output the dirstack after popping off it. The **–l**, **–n**, **-v**, and **-p** options have the same meaning as for the **dirs** builtin.

printenv [**–hv0**] [*name ...*]
Print the names and values of environment variables identified by each *name*. The **–0** option terminates each entry with NULL instead of a newline character.

pushd [**–chlnpsv**] [*+N | −N | dir*]
Push directories on the stack and **cd** to them. If *N* is positive, it rotates the stack and bring the N-th directory, counting from 0 from the left, to the top of the stack. If it is negative, it rotates the stack and bring the N-th directory, counting from 0 from the right, to the top of the stack. If **dir** is supplied, it is pushed on the stack and **cd** is called to change the working directory to **dir**. If **dir** is dash **–**, this equals the previous working directory, as stored in the **$PWD** variable. The **–c** option manipulates the stack without **cd**ing to the directory. The **–s** option suppresses the output the dirstack after popping off it. The **–l**, **–n**, **-v**, and **-p** options have the same meaning as for the **dirs** builtin.

If called without arguments, **pushd** exchanges the top two directories on the stack and calls **cd** to change the current working directory to the new top directory. If the **pushdtohome** extra option is set (by calling `setx -s pushdtohome`), **pushd** pushes the value of **$HOME** and **cd**´s to it instead of exchanging the top two directories. If the **dunique** extra option is set, **pushd** removes instances of **dir** from the stack before pushing it. If the **dextract** extra option is set, **pushd** extracts the N-th directory and pushes it on top of the stack.

pwd [**−hv**] [**−L** | **−P**]

Output the value of the current working directory. The **−L** option is the default option, which prints the logical name of the current working directory. If the **−P** option is specified, symbolic links are resolved. If both options are supplied, the last instance of **−L** or **−P** determines which option is used.

read [**−hv**] [**−rs**] [**−d** *delim*] [**−nN** *n*] [**−t** *timeout*] [**−u** *fd*] [**−p** *msg*] [*varname* ...]

Read a line and break it up into fields using **$IFS** characters as field separators. The escape character '\' removes the special meaning for the following character and for line continuation. The **−d** option causes **read** to continue reading input up to the first character of *delim*, rather than '\n'. The **−n** option causes at most *n* bytes to be read, rather than a complete line. The **−N** option causes exactly *n* bytes to be read, unless EOF has been reached, or the read timed out because of the **−t** option. The **−r** causes the '\' character to lose its special meaning as an escape character. The first field is assigned to the first *varname*, the second field to the second *varname*, and so on. Extra fields are assigned to the last *varname*. If the **−s** option is supplied, input is saved as a command in the history list. The **−u** option specifies a one digit file descriptor to read input from. The **−t** option specifies timeout (in seconds) when reading from a terminal or pipe. If *varname* is omitted, **$REPLY** is used as the default variable name. The **-p** option prints the string *msg* before reading input. The exit status is 0 unless EOF is encountered or read timed out.

readonly [**−p**] [*name*[=*value*]] ...

If **name** is not specified, the names and values of readonly variables are printed with the values quoted to allow reinput to the shell. The **−p** option causes the word **readonly** to be inserted before each name. Otherwise, the specified names are marked as readonly.

repeat [**−hv**] *count command*

Repeat executing *command* for *count* times.

return [*n*]

Return from a shell function or script with the exit status *n*. The value returned is the least significant 8 bits of *n*. If *n* is omitted, the return status is that of the last command executed. If **return** is invoked while not in a function or script, it behaves in the same way as **exit**.

set [**-BCEGHTabdefhkmnoprstuvx**] [**-o** [*option*] ...] [*arg* ...]

set [**+BCEGHTabdefhkmnoprstuvx**] [**+o** [*option*] ...] [*arg* ...]

Set or unset shell options. This utility supports the following options:

−a All subsequent variables that are defined are marked for export.

−b Print job completion messages as soon as a background job changes state.

−B Enable brace pattern field generation. This is set by default.

−C Prevent output redirection operators from truncating existing files. Files created are opened with the **O_EXCL** mode. **>|** must be used in order to truncate a file when this option is set.

−d Dump the parser's Abstract Syntax Tree (AST) before executing commands.

−e If a command exits with non-zero exit status, execute the **ERR** trap, if set, and exit the shell.

−E **ERR** traps are inherited by shell functions, command substitutions and subshells.

−f Disable file name generation.

−h Hash all command names on first encounter.

−H Enable history substitution.

−k All variable assignments are placed in the environment for a command, not just those that precede the command name.

−m Turn on job control. Background jobs are run in separate process groups. Exit status of background jobs is reported in a message. This option is set automatically for interactive shells.

−n Read commands but don't execute them. Ignored by interactive shells.

−o If no *option* is supplied, print the list of options and their current settings to stdout. When invoked as '+o', options are printed in a format that can be reinput to the shell to restore the settings.

The following options can be passed to **−o**:

allexport	Same as **−a**.
braceexpand	Same as **−B**.
errexit	Same as **−e**.
errtrace	Same as **−E**.
functrace	Same as **−T**.
hashall	Same as **−h**.
hashexpand	Same as **−H**.
history	Same as **−w**.
ignoreeof	The shell does not exit on **EOF**.
keyword	Same as **−k**.
monitor	Same as **−m**.
noclobber	Same as **−C**.
noexec	Same as **−n**.
noglob	Same as **−f**.
nolog	Do not save function definitions in the history file.
notify	Same as **−b**.
nounset	Same as **−u**.
onecmd	Same as **−t**.
pipefail	Pipeline exit status is that of the rightmost command with non-zero exit status.
privileged	Same as **−p**.
verbose	Same as **−v**.
vi	Enter vi insert mode until **ESC** (033) is pressed, where control mode is entered.
xtrace	Same as **−x**.

If no option name is supplied, the current options settings are printed.

−p Enter the privileged mode. Disable processing of the **$HOME/.profile** file. This mode is set if the effective uid (or gid) is not equal to the real uid (or gid). Turning this option off causes the effective uid and gid to be set to the real uid and gid.

−r Enable the restricted shell. This option cannot be unset once set.

−t Exit after reading and executing one command.

−T **DEBUG** and **RETURN** traps are inherited by shell functions, command substitutions and subshells.

−u Treat unset parameters as error when performing variable substitution.

−v Print shell input lines as they are read.

−x Print the command line as commands are executed.

−− Do not change any of the options. Useful when setting $1 to a value that begins with '-'. If no arguments follow this option, positional parameters are unset.

Using '+' instead of '-' causes options to be turned off. These options can be passed to the shell on invocation. The current set of options can be viewed by reading the shell variable **$-**. The remaining arguments are treated as positional parameters and assigned to **$1**, **$2**, and so on. If no arguments are specified, the names and values of all shell variables are printed on stdout.

setenv [**−hv**] [*name*[=*value*] ...]
Set the value of environment variable *name* to *value*, or NULL if no *value* is given. This utility sets both the environment variable and the shell variable with the same @code{name}. If no arguments are given, it prints the names and values of all the set environment variables.

setx [**−hvpsuqo**] *option*

Set and unset optional (extra) shell options. *option* can be any of the following (the name inside brackets is the shell from which the option was taken/based; **int** means interactive shell, while **non-int** means non-interactive shell):

addsuffix - append space to file- and slash to dir-names on tab completion (tcsh)
autocd - dirs passed as single-word commands are passed to **cd** (bash int)
cdable_vars - **cd** arguments can be variable names (bash)
cdable-vars - same as the above
checkhash - for hashed commands, check the file exists before exec'ing (bash)
checkjobs - list stopped/running jobs and warn user before exit (bash int)
checkwinsize - check window size after external cmds, updating **$LINES** and **$COLUMNS** (bash)
clearscreen - clear the screen on shell's startup
cmdhist - save multi-line command in a single history entry (bash)
complete_fullquote - quote metacharacters in filenames during completion (bash)
complete-fullquote - same as the above
dextract - **pushd** extracts the given dir instead of rotating the stack (tcsh)
dotglob - files starting with **.** are included in filename expansion (bash)
dunique - **pushd** removes similar entries before pushing dir on the stack (tcsh)
execfail - failing to **exec** a file doesn't exit the shell (bash non-int)
expand_aliases - perform alias expansion (bash)
expand-aliases - same as the above
extglob - enable ksh-like extended pattern matching (bash)
failglob - failing to match filenames to patterns result in expansion error (bash)
force_fignore - **$FIGNORE** determines which words to ignore on word expansion (bash)
force-fignore - same as the above
globasciiranges - bracket pattern matching expressions use the C locale (bash)
histappend - append (don't overwrite) the history list to **$HISTFILE** (bash)
histreedit - enable the user to re-redit a failed history substitution (bash int)
histverify - reload (instead of directly execute) history substitution results (bash int)
hostcomplete - perform hostname completion for words containing @ (bash int)
huponexit - send **SIGHUP** to all jobs on exit (bash int login)
inherit_errexit - command substitution subshells inherit the **−e** option (bash)
inherit-errexit - same as the above
interactive_comments - recognize **#** as the beginning of a comment (bash int)
interactive-comments - same as the above
lastpipe - last cmd of foreground pipeline is run in the current shell (bash)
lithist - save multi-line commands with embedded newlines (bash with 'cmdhist' on)
listjobs - list jobs when a job changes status (tcsh)
listjobs_long - list jobs (detailed) when a job changes status (tcsh)
listjobs-long - same as the above
localvar_inherit - local vars inherit value/attribs from previous scopes (bash)
localvar-inherit - same as the above
localvar_unset - allow unsetting local vars in previous scopes (bash)
localvar-unset - same as the above
login_shell - indicates a login shell (cannot be changed) (bash)
login-shell - same as the above
mailwarn - warn about mail files that have already been read (bash)
nocaseglob - perform case-insensitive filename expansion (bash)
nocasematch - perform case-insensitive pattern matching (bash)
nullglob - patterns expanding to 0 filenames expand to (bash)
printexitvalue - output non-zero exit status for external commands (tcsh)
progcomp - enable programmable completion (not yet implemented) (bash)
progcomp_alias - allow alias expansion in completions (not yet implemented) (bash)

promptvars - perform word expansion on prompt strings (bash)
pushdtohome - **pushd** without arguments pushed ~ on the stack (tcsh)
recognize_only_executables - only executables are recognized in command completion (tcsh)
recognize-only-executables - same as the above
restricted_shell - indicates a restricted shell (cannot be changed) (bash)
restricted-shell - same as the above
savedirs - save the directory stack when login shell exits (tcsh)
savehist - save the history list when shell exits (tcsh)
shift_verbose - allow the **shift** builtin to output error messages (bash)
shift-verbose - same as the above
sourcepath - the source builtin uses **$PATH** to find files (bash)
usercomplete - perform hostname completion for words starting with ~
xpg_echo - echo expands backslash escape sequences by default (bash)
xpg-echo - same as the above

The **−o** option restricts options to those recognized by `set -o`. The **−p** option prints output that can be re-input to the shell. The **−q** option suppresses normal output. the return status tells whether options are set or not. The **−s** option sets (enables) each passed option. The **−u** option unsets (disables) each passed option.

shift [*n*]
Positional parameters from *$n+1* onwards are renamed *$1* ... . The default value if *n* is omitted is 1. *n* should be a non-negative number less than or equal to **$#**.

source [**−hv**] *file*
Execute commands in the current environment. Commands are read from *file* and then executed in the current execution environment. This command is the same as **dot** or `.`, except when the **−h** option is given, where *file* is read and the commands are added to the history list, which is identical to invoking `history -L`.

stop [**−hv**] *job* ...
Stop each background job specified by *job*. See the *Jobs* section for a description of the format of **job**.

suspend [**−fhv**]
Suspend execution of the shell. The **−f** option forces the suspend, even if the shell is a login shell.

test −option *expression*
Test file attributes and compare strings. For the list of options and their meanings, see *Conditional Expressions*.

times Write process times.

trap [**−hvlp**] [*action*] [*sig*] ...
The **−p** option causes the trap action associated with each specified trap to be printed. Otherwise, *action* is processed as if passed to **eval** when the shell receives the signal *sig*, which can be a signal number or a symbolic name. Signals ingored on shell startup cannot be trapped or reset. If *action* is omitted and the first *sig* is a number, or if *action* is a hyphen '-', traps for each *sig* are reset to their original values. If *action* is a null string, the signal is ignored by the shell and its children. If *sig* is **ERR**, *action* is executed whenever a command returns non-zero exit status. If *sig* is *DEBUG*, *action* is executed before each command is executed. If *sig* is **0** or **EXIT**, *action* is executed on exit from the shell. If no arguments are supplied, **trap** prints the traps associated with each signal number. The **−l** option lists all conditions and their signal numbers. The **−p** option prints the trap actions associated with each *sig*.

type *command* ...
Write a description of **command** type.

true Return true result (zero exit status).

ulimit [**−h**] [**−acdflmnpstuvHS**] [*limit*]
Set or display resource limits. The limit for a specified resource is set when *limit* is specified. The value of *limit* can be numeric, or one of the special values **unlimited**, **soft** or **hard**. When more than one resource is specified, the limit name and its unit are printed before printing the limit's value. If no option is specified, **−f** is assumed.

The following are the available resource limits:

−H Set or display the hard limit for the specified resource.
−S Set or display the soft limit for the specified resource.
−a List all the current resource limits.
−c The size of core dumps (512-byte blocks).
−d The size of the data segment (Kbytes).
−f The max size of files written by the current process (512-byte blocks).
−l The max size of memory a process may lock.
−m The the size of physical memory (Kbytes).
−n The number of file descriptors.
−p The size of pipe buffers (512-byte blocks).
−s The the size of the stack (Kbytes).
−t The number of CPU seconds to be used by each process.
−v The size of virtual memory (Kbytes).

umask [**−hvp**] [**−S**] [*mask*]
The file creation mask (**umask**) is set to *mask*, which can be either an octal number or a symbolic value. If a symbolic value is specified, the new **umask** is the complement of the result of applying the given mask to the complement of the previous **umask**. If *mask* is omitted, the current value of **umask** is printed. The **−S** option prints **umask** as a symbolic value. Otherwise, **umask** is printed as an octal number. The **−p** option prints output that can be reused as shell input.

unalias [**−a**] *name*
Removed the aliases specified by the list of names from the alias list. The **−a** option causes all the aliases to be removed.

unlimit [**−hHfSv**] [*limit ...*]

unlimit [**−HS**] **−a**
Remove limits on system resource *limit*, which can be one of the following:

core, -c the maximum size of core files created
data, -d the maximum size of a process's data segment
nice, -e the maximum nice value (scheduling priority)
file, -f the maximum size of files written by a process
signal, -i the maximum number of pending signals
mlock, -l the maximum size of memory a process may lock
rss, -m the maximum resident set size (RSS)
fd, -n the maximum number of open file descriptors
buffer, -p the pipe buffer size in kbytes
message, -q the maximum number of kbytes in POSIX message queues
rtprio, -r the maximum real-time priority
stack, -s the maximum stack size
cputime, -t the maximum amount of cpu time (seconds)
userproc, -u the maximum number of user processes
virtmem, -v the size of virtual memory
flock, -x the maximum number of file locks
all, -a all the above

Options and limit names must be passed separately. To remove all hard limits, invoke either of the following commands:

unlimit -H -a
unlimit -H all

The **−a** option removes limits on all resources. The **−f** option ignores errors. The **−H** option removes hard limits (only root can do this). The **−S** option removes soft limits (the default).

unsetenv [**−hv**] [*name ...*]
Unset environment variable values. This utility unsets both the environment variable and the shell variable with the same *name*. If no arguments are given, nothing is done.

unset [**−fv**] *varname*
Unset the variables specified by the list of **varname**s. Readonly variables cannot be unset. If the **−f** option is supplied, the names are treated as function names. If the **−v** option is supplied, the names are treated as variable names. The default option is **−v**.

wait [**−hfnv**] [*job ...*]
Wait for the specified job or process and report its termination status. If *job* is not specified, all active child processes are waited for. The exit status is that of the last process waited for if *job* is specified, otherwise it is zero. See the *Jobs* section for a description of the format of *job*.

The **−f** option forces jobs/processes to exit. The **−n** option waits for any job or process.

whence [**−afhpv**] *name ...*
For each *name*, indicate how it would be interpreted if it was used as a command to the shell. The **−v** option produces a more verbose output. The **−f** option doesn't search for functions. The **−p** option does a path search for each *name* even if it was an alias, a function, or a reserved word. The **−a** option is similar to **−v**, but causes all interpretations of the specified name to be printed.

SHELL INVOCATION

If upon invocation, the first character of argument zero is a hyphen '-', the shell is a login shell and commands are read from */etc/profile* and then from *.profile* in the current working directory or, if this file is not found, from *$HOME/.profile*.

Interactive shells read */etc/lshrc*, followed by the file named by the result of parameter expansion, command substitution, and arithmetic expansion of the value of the environment variable **$ENV** if the file exists and is readable. If the **−s** option is not supplied and an argument is given which is an existing file, the shell reads and executes that file as a shell script.

Commands are then read and executed. The following options are recognized by the shell upon startup:

−c If this option is supplied, commands are read from the first argument after the **−c** option. Any remaining arguments become positional parameters, starting from argument number 0.

−i If this option is supplied, or if the shell's stdin and stderr are attached to a terminal device (as indicated by **isatty()**), this shell is interactive. Interactive shells ignore the **TERM** signal (so that **kill 0** does not kill an interactive shell). **INTR** is caught and ignored (so that wait is interruptible). **QUIT** is always ignored by the shell.

−r If this option is supplied, the shell becomes a restricted shell.

−s If this option is supplied, or if no arguments remain, commands are read from stdin.

The remaining options and arguments are recognized as described for the **set** builtin utility.

THE RESTRICTED SHELL

The restricted shell is used to set up and execution environment more controlled than that of the standard shell. The restricted shell behaves similar to the normal shell, except that the following actions are not permitted:

* Unsetting the restricted option
* Changing directory using **cd**
* Setting or unsetting the value or attributes of **$SHELL**, **$ENV** or **$PATH**
* Specifying paths or command names containing '/'
* Redirecting output using '>', '>|', '<>', or '>>'

* Using **command −p** to invoke a command

EXIT STATUS

The shell's exit status can be one of:

non-zero
Returned when errors, such as syntax errors, are detected by the shell. Non-interactive shells exit in this case.

zero If no commands were executed.

Other values
The exit status of last command executed is returned on exit.

FILES

/etc/profile
The system initialization file (executed by a login shell).

/etc/lshrc
The system-wide startup file (executed by an interactive shell).

$HOME/.profile
The personal initialization file (executed by a login shell after executing */etc/profile*).

$HOME/.lshrc
The default personal initialization file (executed after */etc/lshrc* by an interactive shell).

$HOME/.lsh_history
The default personal initialization file (executed after */etc/lshrc*

/dev/null
The NULL device.

/etc/logout
Shell script executed on logout.

$HOME/logout
Shell script executed on logout after */etc/logout*.

/etc/passwd
Default file used to perform ~user substitutions.

FEATURES

- Shell language and interpreter are mostly POSIX-compliant
- Standard POSIX regular and special builtins conforming to POSIX
- Extended features and utilities borrowed from ksh, bash and tcsh
- Well documented source code

TODO

- Add support for Unicode strings and message translation
- Implement more bash-like and ksh-like builtin utilities, such as print and printf
- Ensure the builtin utilities (and the shell program itself) are POSIX-compliant
- Testing, bug reporting, debugging and improving the shell

SEE ALSO

info lsh

AUTHOR

Mohammed Isam <mohammed_isam1984@yahoo.com>