NAME

ash, sh, ., break, case, cd, command, continue, eval, exec, exit, export, for, getopts, hash, if, jobs, local, read, readonly, return, set, setvar, shift, trap, umask, unset, wait, while − a shell

SYNOPSIS

ash [**-efIijnsxz**] [**+efIijnsxz**] [**-c** *command*] [*arg*] ...

COPYRIGHT

DESCRIPTION

Ash is a version of *sh* with features similar to those of the System V shell. This manual page lists all the features of *ash* but concentrates on the ones not in other shells.

Invocation

If the **-c** options is given, then the shell executes the specified shell command. The **-s** flag cause the shell to read commands from the standard input (after executing any command specified with the **-c** option. If neither the **-s** or **-c** options are set, then the first *arg* is taken as the name of a file to read commands from. If this is impossible because there are no arguments following the options, then *ash* will set the **-s** flag and will read commands from the standard input.

The shell sets the initial value of the positional parameters from the *arg*s remaining after any *arg* used as the name of a file of commands is deleted.

The flags (other than **-c**) are set by preceding them with ‘‘-’’ and cleared by preceding them with ‘‘+’’; see the *set* builtin command for a list of flags. If no value is specified for the **-i** flag, the **-s** flag is set, and the standard input and output of the shell are connected to terminals, then the **-i** flag will be set. If no value is specified for the **-j** flag, then the **-j** flag will be set if the **-i** flag is set.

When the shell is invoked with the **-c** option, it is good practice to include the *-i* flag if the command was entered interactively by a user. For compatibility with the System V shell, the *-i* option should come after the **-c** option.

If the first character of argument zero to the shell is ‘‘-’’, the shell is assumed to be a login shell, and the files **/etc/profile** and **.profile** are read if they exist. If the environment variable SHINIT is set on entry to the shell, the commands in SHINIT are normally parsed and executed. SHINIT is not examined if the shell is a login shell, or if it the shell is running a shell procedure. (A shell is considered to be running a shell procedure if neither the **-s** nor the **-c** options are set.)

Control Structures

A *list* is a sequence of zero or more commands separated by newlines, semicolons, or ampersands, and optionally terminated by one of these three characters. (This differs from the System V shell, which requires a list to contain at least one command in most cases.) The commands in a list are executed in the order they are written. If command is followed by an ampersand, the shell starts the command and immediately proceed onto the next command; otherwise it waits for the command to terminate before proceeding to the next one.

‘‘&&’’ and ‘‘||’’ are binary operators. ‘‘&&’’ executes the first command, and then executes the second command iff the exit status of the first command is zero. ‘‘||’’ is similar, but executes the second command iff the exit status of the first command is nonzero. ‘‘&&’’ and ‘‘||’’ both have the same priority.

The ‘‘|’’ operator is a binary operator which feeds the standard output of the first command into the standard input of the second command. The exit status of the ‘‘|’’ operator is the exit status of the second command. ‘‘|’’ has a higher priority than ‘‘||’’ or ‘‘&&’’.

An *if* command looks like

```
  if list
  then    list
[ elif list
```

then list] ...
[**else** list]
fi

A *while* command looks like

while list
do list
done

The two lists are executed repeatedly while the exit status of the first list is zero. The *until* command is similar, but has the word **until** in place of **while**
repeats until the exit status of the first list is zero.

The *for* command looks like

for variable **in** word...
do list
done

The words are expanded, and then the list is executed repeatedly with the variable set to each word in turn. **do** and **done** may be replaced with ‘‘{’’ and ‘‘}’’.

The *break* and *continue* commands look like

break [num]
continue [num]

Break terminates the *num* innermost *for* or *while* loops. *Continue* continues with the next iteration of the *num'th* innermost loop. These are implemented as builtin commands.

The *case* command looks like

case word **in**
pattern**)** list **;;**
...
esac

The pattern can actually be one or more patterns (see *Patterns* below), separated by ‘‘|’’ characters.

Commands may be grouped by writing either

(list**)**

or

{ list; **}**

The first of these executes the commands in a subshell.

A function definition looks like

name **( )** command

A function definition is an executable statement; when executed it installs a function named **name** and returns an exit status of zero. The command is normally a list enclosed between ‘‘{’’ and ‘‘}’’.

Variables may be declared to be local to a function by using a *local* command. This should appear as the first staement of a function, and looks like

> **local** [variable | **-**] ...

Local is implemented as a builtin command.

When a variable is made local, it inherits the initial value and exported and readonly flags from the variable with the same name in the surrounding scope, if there is one. Otherwise, the variable is initially unset. *Ash* uses dynamic scoping, so that if you make the variable **x** local to function *f*, which then calls function *g*, references to the variable **x** made inside *g* will refer to the variable **x** declared inside *f*, not to the global variable named **x**.

The only special parameter than can be made local is ‘‘**-**’’. Making ‘‘**-**’’ local any shell options that are changed via the *set* command inside the function to be restored to their original values when the function returns.

The *return* command looks like

> **return** [exitstatus]

It terminates the currently executing function. *Return* is implemented as a builtin command.

Simple Commands

A simple command is a sequence of words. The execution of a simple command proceeds as follows. First, the leading words of the form ‘‘name=value’’ are stripped off and assigned to the environment of the command. Second, the words are expanded. Third, the first remaining word is taken as the command name that command is located. Fourth, any redirections are performed. Fifth, the command is executed. We look at these operations in reverse order.

The execution of the command varies with the type of command. There are three types of commands: shell functions, builtin commands, and normal programs.

When a shell function is executed, all of the shell positional parameters (except $0, which remains unchanged) are set to the parameters to the shell function. The variables which are explicitly placed in the environment of the command (by placing assignments to them before the function name) are made local to the function and are set to values given. Then the command given in the function definition is executed. The positional parameters are restored to their original values when the command completes.

Shell builtins are executed internally to the shell, without spawning a new process.

When a normal program is executed, the shell runs the program, passing the parameters and the environment to the program. If the program is a shell procedure, the shell will interpret the program in a subshell. The shell will reinitialize itself in this case, so that the effect will be as if a new shell had been invoked to handle the shell procedure, except that the location of commands located in the parent shell will be remembered by the child. If the program is a file beginning with ‘‘#!’’, the remainder of the first line specifies an interpreter for the program. The shell (or the operating system, under Berkeley UNIX) will run the interpreter in this case. The arguments to the interpreter will consist of any arguments given on the first line of the program, followed by the name of the program, followed by the arguments passed to the program.

Redirection

Input/output redirections can be intermixed with the words in a simple command and can be placed following any of the other commands. When redirection occurs, the shell saves the old values of the file descriptors and restores them when the command completes. The ‘‘<’’, ‘‘>’’, and ‘‘>>’’ redirections open a file for input, output, and appending, respectively. The ‘‘<&digit’’ and ‘‘>&digit’’ makes the input or output a duplicate of the file descriptor numbered by the digit. If a minus sign is used in place of a digit, the standard input or standard output are closed.

The ‘‘<< word’’ redirection takes input from a *here* document. As the shell encounters ‘‘<<’’ redirections, it collects them. The next time it encounters an unescaped newline, it reads the documents in turn. The word following the ‘‘<<’’ specifies the contents of the line that terminates the document. If none of the quoting methods (’’, "", or \) are used to enter the word, then the document is treated like a word inside double quotes: ‘‘$’’ and backquote are expanded and backslash can be used to escape these and to continue long lines. The word cannot contain any variable or command substitutions, and its length (after quoting) must be in the range of 1 to 79 characters. If ‘‘<<-’’ is used in place of ‘‘<<’’, then leading tabs are deleted from the lines of the document. (This is to allow you do indent shell procedures containing here documents in a natural fashion.)

Any of the preceding redirection operators may be preceded by a single digit specifying the file descriptor to be redirected. There cannot be any white space between the digit and the redirection operator.

Path Search

When locating a command, the shell first looks to see if it has a shell function by that name. Then, if PATH does not contain an entry for "%builtin", it looks for a builtin command by that name. Finally, it searches each entry in PATH in turn for the command.

The value of the PATH variable should be a series of entries separated by colons. Each entry consists of a directory name, or a directory name followed by a flag beginning with a percent sign. The current directory should be indicated by an empty directory name.

If no percent sign is present, then the entry causes the shell to search for the command in the specified directory. If the flag is ‘‘%builtin’’ then the list of shell builtin commands is searched. If the flag is ‘‘%func’’ then the directory is searched for a file which is read as input to the shell. This file should define a function whose name is the name of the command being searched for.

Command names containing a slash are simply executed without performing any of the above searches.

The Environment

The environment of a command is a set of name/value pairs. When the shell is invoked, it reads these names and values, sets the shell variables with these names to the corresponding values, and marks the variables as exported. The *export* command can be used to mark additional variables as exported.

The environment of a command is constructed by constructing name/value pairs from all the exported shell variables, and then modifying this set by the assignments which precede the command, if any.

Expansion

The process of evaluating words when a shell procedure is executed is called *expansion*. Expansion consists of four steps: variable substitution, command substitution, word splitting, and file name generation. If a word is the expression following the word **case** in a case statement, the file name which follows a redirection symbol, or an assignment to the environment of a command, then the word cannot be split into multiple words. In these cases, the last two steps of the expansion process are omitted.

Variable Substitution

To be written.

Command Substitution

Ash accepts two syntaxes for command substitution:

‘*list*‘

and

$(*list*)

Either of these may be included in a word. During the command substitution process, the command (syntactly a *list*) will be executed and anything that the command writes to the standard output will be captured by the shell. The final newline (if any) of the output will be deleted; the rest of the output will be substituted for the command in the word.

Word Splitting

When the value of a variable or the output of a command is substituted, the resulting text is subject to word splitting, unless the dollar sign introducing the variable or backquotes containing the text were enclosed in double quotes. In addition, ‘‘$@’’ is subject to a special type of splitting, even in the presence of double quotes.

Ash uses two different splitting algorithms. The normal approach, which is intended for splitting text separated by which space, is used if the first character of the shell variable IFS is a space. Otherwise an alternative experimental algorithm, which is useful for splitting (possibly empty) fields separated by a separator character, is used.

When performing splitting, the shell scans the replacement text looking for a character (when IFS does not begin with a space) or a sequence of characters (when IFS does begin with a space), deletes the character or sequence of characters, and spits the word into two strings at that point. When IFS begins with a space, the shell deletes either of the strings if they are null. As a special case, if the word containing the replacement text is the null string, the word is deleted.

The variable ‘‘$@’’ is special in two ways. First, splitting takes place between the positional parameters, even if the text is enclosed in double quotes. Second, if the word containing the replacement text is the null string and there are no positional parameters, then the word is deleted. The result of these rules is that "$@" is equivalent to "$1" "$2" ... "$*n*", where *n* is the number of positional parameters. (Note that this differs from the System V shell. The System V documentation claims that "$@" behaves this way; in fact on the System V shell "$@" is equivalent to "" when there are no positional paramteters.)

File Name Generation

Unless the **-f** flag is set, file name generation is performed after word splitting is complete. Each word is viewed as a series of patterns, separated by slashes. The process of expansion replaces the word with the names of all existing files whose names can be formed by replacing each pattern with a string that matches the specified pattern. There are two restrictions on this: first, a pattern cannot match a string containing a slash, and second, a pattern cannot match a string starting with a period unless the first character of the pattern is a period.

If a word fails to match any files and the **-z** flag is not set, then the word will be left unchanged (except that the meta-characters will be converted to normal characters). If the **-z** flag is set, then the word is only left unchanged if none of the patterns contain a character that can match anything besides itself. Otherwise the **-z** flag forces the word to be replaced with the names of the files that it matches, even if there are zero names.

Patterns

A *pattern* consists of normal characters, which match themselves, and meta-characters. The meta-characters are ‘‘!’’, ‘‘*’’, ‘‘?’’, and ‘‘[’’. These characters lose there special meanings if they are quoted. When command or variable substitution is performed and the dollar sign or back quotes are not double quoted, the value of the variable or the output of the command is scanned for these characters and they are turned into meta-characters.

Two exclamation points at the beginning of a pattern function as a ‘‘not’’ operator, causing the pattern to match any string that the remainder of the pattern does *not* match. Other occurances of exclamation points in a pattern match exclamation points. Two exclamation points are required rather than one to decrease the incompatibility with the System V shell (which does not treat exclamation points specially).

An asterisk (‘‘*’’) matches any string of characters. A question mark matches any single character. A left bracket (‘‘[’’) introduces a character class. The end of the character class is indicated by a ‘‘]’’; if the ‘‘]’’ is missing then the ‘‘[’’ matches a ‘‘[’’ rather than introducing a character class. A character class matches any of the characters between the square brackets. A range of characters may be specified using a minus sign. The character class may be complemented by making an exclamation point the first character of the character class.

To include a ‘‘]’’ in a character class, make it the first character listed (after the ‘‘!’’, if any). To include a minus sign, make it the first or last character listed.

The /u Directory

By convention, the name ‘‘/u/user’’ refers to the home directory of the specified user. There are good reasons why this feature should be supported by the file system (using a feature such as symbolic links) rather than by the shell, but *ash* is capable of performing this mapping if the file system doesn’t. If the mapping is done by *ash*, setting the **-f** flag will turn it off.

Character Set

Ash silently discards nul characters. Any other character will be handled correctly by *ash*, including characters with the high order bit set.

Job Names and Job Control

The term *job* refers to a process created by a shell command, or in the case of a pipeline, to the set of processes in the pipeline. The ways to refer to a job are:

%*number*
%*string*
%%
process_id

The first form identifies a job by job number. When a command is run, *ash* assigns it a job number (the lowest unused number is assigned). The second form identifies a job by giving a prefix of the command used to create the job. The prefix must be unique. If there is only one job, then the null prefix will identify the job, so you can refer to the job by writing ‘‘%’’. The third form refers to the *current job*. The current job is the last job to be stopped while it was in the foreground. (See the next paragraph.) The last form identifies a job by giving the process id of the last process in the job.

If the operating system that *ash* is running on supports job control, *ash* will allow you to use it. In this case, typing the suspend character (typically ˆZ) while running a command will return you to *ash* and will make the suspended command the current job. You can then continue the job in the background by typing *bg*, or you can continue it in the foreground by typing *fg*.

Atty

If the shell variable ATTY is set, and the shell variable TERM is not set to ‘‘emacs’’, then *ash* generates appropriate escape sequences to talk to *atty*(1).

Exit Statuses

By tradition, an exit status of zero means that a command has succeeded and a nonzero exit status indicates that the command failed. This is better than no convention at all, but in practice it is extremely useful to allow commands that succeed to use the exit status to return information to the caller. A variety of better conventions have been proposed, but none of them has met with universal approval. The convention used by *ash* and all the programs included in the *ash* distribution is as follows:

0	Success.
1	Alternate success.
2	Failure.
129-...	Command terminated by a signal.

The *alternate success* return is used by commands to indicate various conditions which are not errors but which can, with a little imagination, be conceived of as less successful than plain success. For example, *test* returns 1 when the tested condition is false and *getopts* returns 1 when there are no more options. Because this convention is not used universally, the **-e** option of *ash* causes the shell to exit when a command returns 1 even though that contradicts the convention described here.

When a command is terminated by a signal, the uses 128 plus the signal number as the exit code for the command.

Builtin Commands

This concluding section lists the builtin commands which are builtin because they need to perform some operation that can't be performed by a separate process. In addition to these, there are several other commands (*catf*, *echo*, *expr*, *line*, *nlecho*, *test*, ‘‘*:*’’, and *true*) which can optionally be compiled into the shell. The builtin commands described below that accept options use the System V Release 2 *getopt*(3) syntax.

bg [*job*] ...
: Continue the specified jobs (or the current job if no jobs are given) in the background. This command is only available on systems with Bekeley job control.

command *command arg...*
: Execute the specified builtin command. (This is useful when you have a shell function with the same name as a builtin command.)

cd [*directory*]
: Switch to the specified directory (default $HOME). If the an entry for CDPATH appears in the environment of the cd command or the shell variable CDPATH is set and the directory name does not begin with a slash, then the directories listed in CDPATH will be searched for the specified directory. The format of CDPATH is the same as that of PATH. In an interactive shell, the cd command will print out the name of the directory that it actually switched to if this is different from the name that the user gave. These may be different either because the CDPATH mechanism was used or because a symbolic link was crossed.

. *file*
: The commands in the specified file are read and executed by the shell. A path search is not done to find the file because the directories in PATH generally contain files that are intended to be executed, not read.

eval *string...*
: The strings are parsed as shell commands and executed. (This differs from the System V shell, which concatenates the arguments (separated by spaces) and parses the result as a single command.)

exec [*command arg...*]
: Unless *command* is omitted, the shell process is replaced with the specified program (which must be a real program, not a shell builtin or function). Any redirections on the exec command are marked as permanent, so that they are not undone when the exec command finishes. If the command is not found, the exec command causes the shell to exit.

exit [*exitstatus*]
: Terminate the shell process. If *exitstatus* is given it is used as the exit status of the shell; otherwise the exit status of the preceding command is used.

export *name...*
: The specified names are exported so that they will appear in the environment of subsequent commands. The only way to un-export a variable is to unset it. *Ash* allows the value of a variable to be set at the same time it is exported by writing

  export name=value

With no arguments the export command lists the names of all exported variables.

fg [*job*]
Move the specified job or the current job to the foreground. This command is only available on systems with Bekeley job control.

getopts *optstring var*
The System V *getopts* command.

hash -rv *command*...
The shell maintains a hash table which remembers the locations of commands. With no arguments whatsoever, the hash command prints out the contents of this table. Entries which have not been looked at since the last *cd* command are marked with an asterisk; it is possible for these entries to be invalid.

With arguments, the hash command removes the specified commands from the hash table (unless they are functions) and then locates them. With the **-v** option, *hash* prints the locations of the commands as it finds them. The **-r** option causes the *hash* command to delete all the entries in the hash table except for functions.

jobid [*job*]
Print the process id's of the processes in the job. If the job argument is omitted, use the current job.

jobs
This command lists out all the background processes which are children of the current shell process.

pwd
Print the current directory. The builtin command may differ from the program of the same name because the builtin command remembers what the current directory is rather than recomputing it each time. This makes it faster. However, if the current directory is renamed, the builtin version of pwd will continue to print the old name for the directory.

read [**-p** *prompt*] [**-e**] *variable*...
The prompt is printed if the **-p** option is specified and the standard input is a terminal. Then a line is read from the standard input. The trailing newline is deleted from the line and the line is split as described in the section on word splitting above, and the pieces are assigned to the variables in order. If there are more pieces than variables, the remaining pieces (along with the characters in IFS that separated them) are assigned to the last variable. If there are more variables than pieces, the remaining variables are assigned the null string.

The **-e** option causes any backslashes in the input to be treated specially. If a backslash is followed by a newline, the backslash and the newline will be deleted. If a backslash is followed by any other character, the backslash will be deleted and the following character will be treated as though it were not in IFS, even if it is.

readonly *name*...
The specified names are marked as read only, so that they cannot be subsequently modified or unset. *Ash* allows the value of a variable to be set at the same time it is marked read only by writing

> **readonly** name=value

With no arguments the readonly command lists the names of all read only variables.

set [{ -*options* | +*options* | **--** }] *arg*...
The *set* command performs three different functions.

With no arguments, it lists the values of all shell variables.

If options are given, it sets the specified option flags, or clears them if the option flags are introduced with a **+** rather than a **-**. Only the first argument to *set* can contain options. The possible options are:

-e Causes the shell to exit when a command terminates with a nonzero exit status, except when the exit status of the command is explicitly tested. The exit status of a command is considered to be explicitly tested if the command is used to control an *if*, *elif*, *while*, or *until*; or if the command is the left hand operand of an ‘‘&&’’ or ‘‘||’’ operator.

-f Turn off file name generation.

-I Cause the shell to ignore end of file conditions. (This doesn’t apply when the shell a script sourced using the ‘‘.’’ command.) The shell will in fact exit if it gets 50 eof’s in a row.

-i Make the shell interactive. This causes the shell to prompt for input, to trap interrupts, to ignore quit and terminate signals, and to return to the main command loop rather than exiting on error.

-j Turns on Berkeley job control, on systems that support it. When the shell starts up, the **-j** is set by default if the **-i** flag is set.

-n Causes the shell to read commands but not execute them. (This is marginally useful for checking the syntax of scripts.)

-s If this flag is set when the shell starts up, the shell reads commands from its standard input. The shell doesn’t examine the value of this flag any other time.

-x If this flag is set, the shell will print out each command before executing it.

-z If this flag is set, the file name generation process may generate zero files. If it is not set, then a pattern which does not match any files will be replaced by a quoted version of the pattern.

The third use of the set command is to set the values of the shell’s positional parameters to the specified *args*. To change the positional parameters without changing any options, use ‘‘**--**’’ as the first argument to *set*. If no args are present, the set command will leave the value of the positional parameters unchanged, so to set the positional parameters to set of values that may be empty, execute the command

```
shift $#
```

first to clear out the old values of the positional parameters.

setvar *variable value*
Assigns *value* to *variable*. (In general it is better to write *variable=value* rather than using *setvar*. *Setvar* is intended to be used in functions that assign values to variables whose names are passed as parameters.)

shift [*n*]
Shift the positional parameters *n* times. A shift sets the value of $1 to the value of $2, the value of $2 to the value of $3, and so on, decreasing the value of $# by one. If there are zero positional parameters, shifting doesn’t do anything.

trap [*action*] *signal*...
Cause the shell to parse and execute *action* when any of the specified signals are received. The signals are specified by signal number. *Action* may be null or omitted; the former causes the specified signal to be ignored and the latter causes the default action to be taken. When the shell forks off a subshell, it resets trapped (but not ignored) signals to the default action. The trap command has no effect on signals that were ignored on entry to the shell.

umask [*mask*]
Set the value of umask (see *umask*(2)) to the specified octal value. If the argument is omitted, the umask value is printed.

unset *name*...
The specified variables and functions are unset and unexported. If a given name corresponds to both a variable and a function, both the variable and the function are unset.

wait [*job*]
Wait for the specified job to complete and return the exit status of the last process in the job. If the argument is omitted, wait for all jobs to complete and the return an exit status of zero.

EXAMPLES

The following function redefines the *cd* command:

```
cd() {
        if command cd "$@"
        thenif test -f .enter
        then. .enter
        elsereturn 0
        fi
        fi
}
```

This function causes the file ‘‘.enter’’ to be read when you enter a directory, if it exists. The *command* command is used to access the real *cd* command. The ‘‘return 0’’ ensures that the function will return an exit status of zero if it successfully changes to a directory that does not contain a ‘‘.enter’’ file. Redefining existing commands is not always a good idea, but this example shows that you can do it if you want to.

The suspend function distributed with *ash* looks like

```
# Copyright (C) 1989 by Kenneth Almquist.  All rights reserved.
# This file is part of ash, which is distributed under the terms
# specified by the Ash General Public License.

suspend() {
        local -
        set +j
        kill -TSTP 0
}
```

This turns off job control and then sends a stop signal to the current process group, which suspends the shell. (When job control is turned on, the shell ignores the TSTP signal.) Job control will be turned back on when the function returns because ‘‘-’’ is local to the function. As an example of what *not* to do, consider an earlier version of *suspend*:

```
suspend() {
        suspend_flag=$-
        set +j
        kill -TSTP 0
        set -$suspend_flag
}
```

There are two problems with this. First, **suspend_flag** is a global variable rather than a local one, which will cause problems in the (unlikely) circumstance that the user is using that variable for some other purpose. Second, consider what happens if shell received an interrupt signal after it executes the first *set* command but before it executes the second one. The interrupt signal will abort the shell function, so that the second *set* command will never be executed and job control will be left off. The first version of *suspend* avoids this problem by turning job control off only in a local copy of the shell options. The local copy of the shell options is discarded when the function is terminated, no matter how it is terminated.

HINTS

Shell variables can be used to provide abbreviations for things which you type frequently. For example, I set

```
export h=$HOME
```

in my .profile so that I can type the name of my home directory simply by typing ‘‘$h’’.

When writing shell procedures, try not to make assumptions about what is imported from the environment. Explicitly unset or initialize all variables, rather than assuming they will be unset. If you use cd, it is a good idea to unset CDPATH.

People sometimes use ‘‘<&-’’ or ‘‘>&-’’ to provide no input to a command or to discard the output of a command. A better way to do this is to redirect the input or output of the command to **/dev/null**.

Word splitting and file name generation are performed by default, and you have to explicitly use double quotes to suppress it. This is backwards, but you can learn to live with it. Just get in the habit of writing double quotes around variable and command substitutions, and omit them only when you really want word splitting and file name generation. If you want word splitting but not file name generation, use the **-f** option.

AUTHORS

Kenneth Almquist

SEE ALSO

echo(1), expr(1), line(1), pwd(1), true(1).

BUGS

When command substitution occurs inside a here document, the commands inside the here document are run with their standard input closed. For example, the following will not word because the standard input of the *line* command will be closed when the command is run:

```
cat <<-!
Line 1: $(line)
Line 2: $(line)
!
```

Unsetting a function which is currently being executed may cause strange behavior.

The shell syntax allows a here document to be terminated by an end of file as well as by a line containing the terminator word which follows the ‘‘<<’’. What this means is that if you mistype the terminator line, the shell will silently swallow up the rest of your shell script and stick it in the here document.