IEEE P1003.2 Draft 11.2 – September 1991

Utilities other than the special built-ins (see 3.14) shall be invoked in a separate environment that consists of the following. The initial value of these objects shall be the same as that for the parent shell, except as noted below.

— Open files inherited on invocation of the shell, open files controlled by the `exec` special built-in (see 3.14.6), plus any modifications and additions specified by any redirections to the utility.

— Current working directory.

— File creation mask.

— If the utility is a shell script, traps caught by the shell shall be set to the default values and traps ignored by the shell shall be set to be ignored by the utility. If the utility is not a shell script, the trap actions (default or ignore) shall be mapped into the appropriate signal handling actions for the utility.

— Variables with the `export` attribute, along with those explicitly exported for the duration of the command, shall be passed to the utility as POSIX.1 {8} environment variables.

The environment of the shell process shall not be changed by the utility unless explicitly specified by the utility description (for example, `cd` and `umask`).

A subshell environment shall be created as a duplicate of the shell environment, except that signal traps set by that shell environment shall be set to the default values. Changes made to the subshell environment shall not affect the shell environment. Command substitution, commands that are grouped with parentheses, and asynchronous lists shall be executed in a subshell environment. Additionally, each command of a multicommand pipeline is in a subshell environment; as an extension, however, any or all commands in a pipeline may be executed in the current environment. All other commands shall be executed in the current shell environment.

3.12.0.1 Shell Execution Environment Rationale. *(This subclause is not a part of P1003.2)*

Some systems have implemented the last stage of a pipeline in the current environment so that commands such as

command `| read foo`

set variable `foo` in the current environment. It was decided to allow this extension, but not require it; therefore, a shell programmer should consider a pipeline to be in a subshell environment, but not depend on it.

The previous description of execution environment failed to mention that each command in a multiple command pipeline could be in a subshell execution environment. For compatibility with some existing shells, the wording was phrased to allow an implementation to place any or all commands of a pipeline in the current environment. However, this means that a POSIX application must assume each command is in a subshell environment, but not depend on it.

The wording about shell scripts is meant to convey the fact that describing ‘‘trap actions’’ can only be understood in the context of the shell command language. Outside this context, such as in a C-language program, signals are the operative condition, not traps.

3.13 Pattern Matching Notation

The pattern matching notation described in this clause is used to specify patterns for matching strings in the shell. Historically, pattern matching notation is related to, but slightly different from, the regular expression notation described in 2.8. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation.

3.13.0.1 Pattern Matching Notation Rationale. *(This subclause is not a part of P1003.2)*

Pattern matching is a simpler concept and has a simpler syntax than regular expressions, as the former is generally used for the manipulation of file names, which are relatively simple collections of characters, while the latter is generally used to manipulate arbitrary text strings of potentially greater complexity. However, some of the basic concepts are the same, so this clause points liberally to the detailed descriptions in 2.8.

3.13.1 Patterns Matching a Single Character

The following *patterns matching a single-character* match a single character: *ordinary characters* , *special pattern characters* , and *pattern bracket expressions* . The pattern bracket expression also shall match a single collating element.

An ordinary character is a pattern that shall match itself. It can be any character
in the supported character set except for NUL, those special shell characters in 3.2 1
that require quoting, and the following three special pattern characters. Match- 1
ing shall be based on the bit pattern used for encoding the character, not on the 1
graphic representation of the character. If any character (ordinary, shell special, 1
or pattern special) is quoted, that pattern shall match the character itself. The 1
shell special characters always require quoting. 1

When unquoted and outside a bracket expression, the following three characters 1
shall have special meaning in the specification of patterns: 1

? A question-mark is a pattern that shall match any character.

* An asterisk is a pattern that shall match multiple characters, as described in 3.13.2.

[The open bracket shall introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in 2.8.3.2 also shall apply to the pattern bracket expression, except that the exclamation-mark character (!) shall replace the circumflex character (^) in its role in a *nonmatching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

When pattern matching is used where shell quote removal is not performed [such as in the argument to the `find –name` primary when `find` is being called using an *exec* function, or in the *pattern* argument to the *fnmatch* () function], special characters can be escaped to remove their special meaning by preceding them with a `<backslash>`. This escaping `<backslash>` shall be discarded. The sequence \\ shall represent one literal backslash. All of the requirements and effects of quoting on ordinary, shell special, and special pattern characters shall apply to escaping in this context.

3.13.1.1 Patterns Matching a Single Character Rationale. *(This subclause is not a part of P1003.2)*

Both ‘‘quoting’’ and ‘‘escaping’’ are described here because pattern matching must work in three separate circumstances:

— Calling directly upon the shell, such as in pathname expansion or in a `case` statement. All of the following will match the string or file `abc`: `abc`, `"abc"`, `a"b"c`, `a\bc`, `a[b]c`, `a["b"]c`, `a[\b]c`, `a?c`, `a*c`. The following will not: `"a?c"`, `a\*c`, `a\[b]c`, `a["\b"]c`.

— Calling a utility or function without going through a shell, as described for `find` and *fnmatch* ().

— Calling utilities such as `find` or `pax` through the shell command line. (Although `find` and `pax` are the only instances of this in the standard utilities, describing it globally here is useful for future utilities that may use pattern matching internally.) In this case, shell quote removal is performed before the utility sees the argument. For example, in

```
find /bin -name "e\c[\h]o" -print
```

after quote removal, the backslashes are presented to `find` and it treats them as escape characters. Both precede ordinary characters, so the `c` and `h` represent themselves and `echo` would be found on many historical systems (that have it in `/bin`). To find a filename that contained shell special characters or pattern characters, both quoting and escaping are required, such as

```
pax -r ... "*a\(\?"
```

to extract a filename ending with ‘`a(?`’.

Conforming applications are required to quote or escape the shell special characters (called ‘‘metacharacters’’ in some historical documentation). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns.

The restriction on circumflex in a bracket expression is to allow implementations that support pattern matching using circumflex as the negation character in addition to the exclamation-mark. 1

3.13.2 Patterns Matching Multiple Characters

The following rules are used to construct *patterns matching multiple characters* from *patterns matching a single character* :

(1) The asterisk (*) is a pattern that shall match any string, including the null string.

(2) The concatenation of *patterns matching a single character* is a valid pattern that shall match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.

(3) The concatenation of one or more *patterns matching a single character* with one or more asterisks is a valid pattern. In such patterns, each asterisk shall match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

3.13.2.1 Patterns Matching Multiple Characters Rationale. *(This subclause is not a part of P1003.2)*

Since each asterisk matches ''zero or more'' occurrences, the patterns `a*b` and `a**b` have identical functionality.

Examples:

`a[bc]`	matches the strings `ab` and `ac`.
`a*d`	matches the strings `ad`, `abd`, and `abcd`, but not the string `abc`.
`a*d*`	matches the strings `ad`, `abcd`, `abcdef`, `aaaad`, and `adddd`;
`*a*d`	matches the strings `ad`, `abcd`, `efabcd`, `aaaad`, and `adddd`.

3.13.3 Patterns Used for Filename Expansion

The rules described so far in 3.13.1 and 3.13.2 are qualified by the following rules that apply when pattern matching notation is used for filename expansion.

(1) The slash character in a pathname shall be explicitly matched by using one or more slashes in the pattern; it cannot be matched by the asterisk or question-mark special characters or by a bracket expression. Slashes in the pattern are identified before bracket expressions; thus, a slash cannot be included in a pattern bracket expression used for filename expansion.

(2) If a filename begins with a period (`.`), the period shall be explicitly matched by using a period as the first character of the pattern or

immediately following a slash character. The leading period shall not be matched by:

— The asterisk or question-mark special characters, or

— A bracket expression containing a nonmatching list (such as `[!a]`), a range expression (such as `[%-0]`), or a character class expression (such as `[[:punct:]]`).

It is unspecified whether an explicit period in a bracket expression matching list (such as `[.abc]`) can match a leading period in a filename.

(3) Specified patterns are matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character requires read permission in the directory containing that component. Any component that does not contain a pattern character requires search permission. For example, given the pattern

```
/foo/bar/x*/bam
```

search permission is needed for directory `/foo`, search and read permissions are needed for directory `bar`, and search permission is needed for each `x*` directory. If the pattern matches any existing filenames or pathnames, the pattern shall be replaced with those filenames and pathnames, sorted according to the collating sequence in effect in the current locale. If the pattern contains an invalid bracket expression or does not match any existing filenames or pathnames, the pattern string shall be left unchanged.

3.13.3.1 Patterns Used for File Name Expansion Rationale. *(This subclause is not a part of P1003.2)*

The caveat about a slash within a bracket expression is derived from historical practice. The pattern `a[b/c]d` will not match such pathnames as `abd` or `a/d`. It will only match a pathname of literally `a[b/c]d`.

Filenames beginning with a period historically have been specially protected from view on UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading period was considered; it is allowed as an implementation extension, but a conforming application cannot make use of it. If this extension becomes popular in the future, it will be considered for a future version of POSIX. 2.

Historical systems have varied in their permissions requirements. To match `f*/bar` has required read permissions on the `f*` directories in the System V shell, but this standard, the C-shell, and KornShell require only search permissions.

3.14 Special Built-in Utilities

The following *special built-in* utilities shall be supported in the shell command language. The output of each command, if any, shall be written to standard output, subject to the normal redirection and piping possible with all commands.

The term *built-in* implies that the shell can execute the utility directly and does not need to search for it. An implementation can choose to make any utility a built-in; however, the special built-in utilities described here differ from regular built-in utilities in two respects:

(1) A syntax error in a special built-in utility may cause a shell executing that utility to abort, while a syntax error in a regular built-in utility shall not cause a shell executing that utility to abort. (See 3.8.1 for the consequences of errors on interactive and noninteractive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value shall be nonzero.

(2) Variable assignments specified with special built-in utilities shall remain
in effect after the built-in completes; this shall not be the case with a reg- 1
ular built-in or other utility. 1

As described in 2.3, the special built-in utilities in this clause need not be provided in a manner accessible via the POSIX.1 {8} *exec* family of functions.

Some of the special built-ins are described as conforming to the utility argument syntax guidelines in 2.10.2. For those that are not, the requirement in 2.11.3 that "--" be recognized as a first argument to be discarded does not apply and a conforming application shall not use that argument.

3.14.1 `break` — Exit from `for`, `while`, or `until` loop

`break` [*n*]

Exit from the smallest enclosing `for`, `while`, or `until` loop, if any; or from the
*n*th enclosing loop if *n* is specified. The value of *n* is an unsigned decimal integer 1
≥ 1. The default is equivalent to *n*=1. If *n* is greater than the number of enclosing loops, the last enclosing loop shall be exited from. Execution continues with the command immediately following the loop.

Exit Status

0 Successful completion. 2

>0 The *n* value was not an unsigned decimal integer ≥ 1. 2

3.14.1.1 `break` Rationale. *(This subclause is not a part of P1003.2)*

Example:

```
for i in *
do
        if test -d "$i"
        then break
        fi
done
```

Consideration was given to expanding the syntax of the `break` and `continue` to refer to a label associated with the appropriate loop, as a preferable alternative to the **[***n***]** method. This new method was proposed late in the development of the standard and adequate consensus could not be formed to include it. However, POSIX.2 does reserve the namespace of command names ending with a colon. It is anticipated that a future implementation could take advantage of this and provide something like:

```
outofloop: for i in a b c d e
do
        for j in 0 1 2 3 4 5 6 7 8 9
        do
                if test -r "${i}${j}"
                then break outofloop
                fi
        done
done
```

and that this might be standardized after implementation experience is achieved.

3.14.2 colon — Null utility

: **[***argument* ...**]**

This utility shall only expand command *argument* s.

Exit Status

Zero.

3.14.2.1 colon Rationale. *(This subclause is not a part of P1003.2)*

The colon (`:`), or null utility, is used when a command is needed, as in the `then` condition of an `if` command, but nothing is to be done by the command.

Example:

```
: ${X=abc}
if      false
then    :
else    echo $X
fi
abc
```

As with any of the special built-ins, the null utility can also have variable assignments and redirections associated with it, such as:

```
x=y : > z
```

which sets variable `x` to the value `y` (so that it persists after the null utility ''completes'') and creates or truncates file `z`.

3.14.3 `continue` — Continue `for`, `while`, or `until` loop

`continue` [*n*]

The `continue` utility shall return to the top of the smallest enclosing `for`, `while`, or `until`, loop, or to the top of the *n*th enclosing loop, if *n* is specified. This involves repeating the condition list of a `while` or `until` loop or performing the next assignment of a `for` loop, and reexecuting the loop if appropriate.

The value of *n* is a decimal integer ≥ 1. The default is equivalent to *n*=`1`. If *n* is greater than the number of enclosing loops, the last enclosing loop is used.

Exit Status

0 Successful completion. 2

>0 The *n* value was not an unsigned decimal integer ≥ 1. 2

3.14.3.1 `continue` Rationale. *(This subclause is not a part of P1003.2)*

Example :

```
for i in *
do
        if test -d "$i"
        then continue
        fi
done
```

3.14.4 `dot` — Execute commands in current environment

`.` *file*

The shell shall execute commands from the *file* in the current environment.

If *file* does not contain a slash, the shell shall use the search path specified by **PATH** to find the directory containing *file*. Unlike normal command search, however, the file searched for by the `dot` utility need not be executable. If no readable file is found, a noninteractive shell shall abort; an interactive shell shall write a diagnostic message to standard error, but this condition shall not be considered a syntax error.

Exit Status

Returns the value of the last command executed, or a zero exit status if no command is executed.

3.14.4.1 dot Rationale. *(This subclause is not a part of P1003.2)*

Some older implementations searched the current directory for the *file*, even if the value of **PATH** disallowed it. This behavior was omitted from POSIX.2 due to concerns about introducing the susceptibility to trojan horses that the user might be trying to avoid by leaving dot out of **PATH**.

The KornShell version of `dot` takes optional arguments that are set to the posi- 1
tional parameters. This is a valid extension that allows a dot script to behave 1
identically to a function.

Example:

```
cat foobar
foo=hello bar=world
. foobar
echo $foo $bar
hello world
```

3.14.5 `eval` — Construct command by concatenating arguments

`eval` [*argument* ...]

The `eval` utility shall construct a command by concatenating *argument* s together, separating each with a `<space>`. The constructed command shall be read and executed by the shell.

Exit Status

If there are no *argument* s, or only null arguments, `eval` shall return a zero exit status; otherwise, it shall return the exit status of the command defined by the string of concatenated *argument* s separated by spaces.

3.14.5.1 `eval` Rationale. *(This subclause is not a part of P1003.2)*

Example:

```
foo=10 x=foo
y='$'$x
echo $y
$foo
eval y='$'$x
echo $y
10
```

3.14.6 `exec` — Execute commands and open, close, and/or copy file descriptors

```
exec [command [argument ...]]
```

The `exec` utility opens, closes, and/or copies file descriptors as specified by any redirections as part of the command.

If `exec` is specified without *command* or *arguments,* and any file descriptors with numbers > 2 are opened with associated redirection statements, it is unspecified whether those file descriptors remain open when the shell invokes another utility.

If `exec` is specified with *command*, it shall replace the shell with *command* without creating a new process. If *argument* s are specified, they are arguments to *command*. Redirection shall affect the current shell execution environment.

Exit Status

If *command* is specified, `exec` shall not return to the shell; rather, the exit status 2
of the process shall be the exit status of the program implementing *command*, 2
which overlaid the shell. If *command* is not found, the exit status shall be 127. If 1
command is found, but it is not an executable utility, the exit status shall be 126. 1
If a redirection error occurs (see 3.8.1), the shell shall exit with a value in the 1
range 1–125. Otherwise, `exec` shall return a zero exit status.

3.14.6.1 `exec` Rationale. *(This subclause is not a part of P1003.2)*

Most historical implementations are not conformant in that

```
foo=bar exec cmd
```

does not pass `foo` to `cmd`.

Earlier drafts stated that ‘‘If specified without *command* or *argument*, the shell sets to close-on-exec file numbers greater than 2 that are opened in this way, so that they will be closed when the shell invokes another program.’’ This was based on the behavior of one version of the KornShell and was made unspecified when it was realized that some existing scripts relied on the more generally historical behavior (leaving all file descriptors open). Furthermore, since the application should have no cognizance of whether a new shell is simply *fork* ()ed, rather than *exec*()ed, it could not consistently rely on the automatic closing behavior anyway. Scripts concerned that child shells could misuse open file descriptors can always close them explicitly, as shown in one of the following examples.

Examples :

Open `readfile` as file descriptor 3 for reading:

```
exec 3< readfile
```

Open `writefile` as file descriptor 4 for writing:

```
exec 4> writefile
```

Make unit 5 a copy of unit 0:

```
exec 5<&0
```

Close file unit 3:

```
exec 3<&–
```

Cat the file `maggie` by replacing the current shell with the `cat` utility:

```
exec cat maggie
```

3.14.7 `exit` — Cause the shell to exit

`exit` [*n*]

The `exit` utility shall cause the shell to exit with the exit status specified by the unsigned decimal integer *n*. If *n* is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

A trap on `EXIT` shall be executed before the shell terminates, except when the `exit` utility is invoked in that trap itself, in which case the shell shall exit immediately.

Exit Status

The exit status shall be *n*, if specified. Otherwise, the value shall be the exit value of the last command executed, or zero if no command was executed. When `exit` is executed in a trap action (see 3.14.13), the ‘‘last command’’ is considered to be the command that executed immediately preceding the trap action.

3.14.7.1 `exit` Rationale. *(This subclause is not a part of P1003.2)*

As explained in other clauses, certain exit status values have been reserved for special uses and should be used by applications only for those purposes:

126	A file to be executed was found, but it was not an executable utility.
127	A utility to be executed was not found.
>128	A command was interrupted by a signal.

Examples:

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```

3.14.8 `export` — Set export attribute for variables

`export` *name* [=*word*]. . .
`export -p`

The shell shall give the export attribute to the variables corresponding to the specified *name* s, which shall cause them to be in the environment of subsequently executed commands.

When `-p` is specified, `export` shall write to the standard output the names and values of all exported variables, in the following format: 1

`"export %s=%s\n"`, <*name*> , <*value*>

The shell shall format the output, including the proper use of quoting, so that it is suitable for re-input to the shell as commands that achieve the same exporting results.

The `export` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

Zero.

3.14.8.1 `export` Rationale. *(This subclause is not a part of P1003.2)*

When no arguments are given, the results are unspecified. Some historical shells use the no-argument case as the functional equivalent of what is required here with `-p`. This feature was left unspecified because it is not existing practice in all shells and some scripts may rely on the now-unspecified results on their implementations. Attempts to specify the `-p` output as the default case were unsuccessful in achieving consensus. The `-p` option was added to allow portable access to the values that can be saved and then later restored using, for instance, a dot script.

Examples:

Export **PWD** and **HOME** variables:

```
export PWD HOME
```

Set and export the **PATH** variable:

```
export PATH=/local/bin:$PATH
```

Save and restore all exported variables:

```
export -p > temp-file
unset a lot of variables
. . . processing
. temp-file
```

3.14.9 `readonly` — Set read-only attribute for variables 1

```
readonly name [=word ]. . .
readonly -p
```

The variables whose *name* s are specified shall be given the `readonly` attribute. The values of variables with the read-only attribute cannot be changed by subsequent assignment, nor can those variables be unset by the `unset` utility.

When `−p` is specified, `readonly` shall write to the standard output the names and values of all read-only variables, in the following format: 1

`"readonly %s=%s\n"`, <*name*> , <*value*>

The shell shall format the output, including the proper use of quoting, so that it is suitable for re-input to the shell as commands that achieve the same attribute-setting results.

The `readonly` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

Zero.

3.14.9.1 `readonly` Rationale. *(This subclause is not a part of P1003.2)*

Example :

```
readonly HOME PWD
```

Some versions of the shell exist that preserve the read-only attribute across separate invocations. POSIX.2 allows this behavior, but does not require it.

See the rationale for `export` (3.14.8.1) for a description of the no-argument and `−p` output cases.

In a previous draft, read-only functions were considered, but they were omitted as not being existing practice or particularly useful. Furthermore, functions must not be `readonly` across invocations to preclude *spoofing* (spoofing is the term for the practice of creating a program that acts like a well-known utility with the intent of subverting the user's real intent) of administrative or security-relevant (or -conscious) shell scripts.

3.14.10 `return` — Return from a function

    return [n]

The return utility shall cause the shell to stop executing the current function or dot script (see 3.14.4). If the shell is not currently executing a function or dot script, the results are unspecified.

Exit Status

The value of the special parameter ? shall be set to *n*, an unsigned decimal integer, or to the exit status of the last command executed if *n* is not specified. If the value of *n* is greater than 255, the results are undefined. When `return` is executed in a trap action (see 3.14.13), the ‘‘last command’’ is considered to be the command that executed immediately preceding the trap action.

3.14.10.1 `return` Rationale. *(This subclause is not a part of P1003.2)*

The behavior of `return` when not in a function or dot script differs between the System V shell and the KornShell. In the System V shell this is an error, whereas in the KornShell, the effect is the same as `exit`.

The results of returning a number greater than 255 are undefined because of differing practices in the various historical implementations. Some shells AND out all but the low order 8 bits; others allow larger values, but not of unlimited size.

See the discussion of appropriate exit status values in 3.14.7.1. 1

3.14.11 `set` — Set/unset options and positional parameters

    set [-aCefnuvx] [argument ...]
    set [+aCefnuvx] [argument ...]
    set -- [argument ...]

Obsolescent version :

    set - [argument ...]

If no options or *argument* s are specified, `set` shall write the names and values of all shell variables in the collation sequence of the current locale. Each *name* shall start on a separate line, using the format:

```
"%s=%s\n", <name>, <value>
```

The *value* string shall be written with appropriate quoting so that it is suitable for re-input to the shell, (re)setting, as far as possible, the variables that are currently set. Readonly variables cannot be reset. See the description of shell quoting in 3.2. 1 1

When options are specified, they shall set or unset attributes of the shell, as described below. When *argument* s are specified, they shall cause positional parameters to be set or unset, as described below. Setting/unsetting attributes and positional parameters are not necessarily related actions, but they can be combined in a single invocation of `set`.

The `set` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that options can be specified with either a leading hyphen (meaning enable the option) or plus-sign (meaning disable it).

The implementation shall support the options in the following list in both their hyphen and plus-sign forms. These options can also be specified as options to `sh`; see 4.56.

`−a` When this option is on, the export attribute shall be set for each variable to which an assignment is performed. (See 3.1.15.) If the assignment precedes a utility name in a command, the export attributes shall not persist in the current execution environment after the utility completes, with the exception that preceding one of the special built-in utilities shall cause the export attribute to persist after the built-in has completed. If the assignment does not precede a utility name in the command, or if the assignment is a result of the operation of the `getopts` or `read` utilities (see 4.27 and 4.52), the export attribute shall persist until the variable is unset.

`−C` (Uppercase C.) Prevent existing files from being overwritten by the shell's `>` redirection operator (see 3.7.2); the `>|` redirection operator shall override this ‘‘noclobber’’ option for an individual file.

`−e` When this option is on, if a simple command fails for any of the reasons listed in 3.8.1 or returns an exit status value >0, and is not part of the compound list following a `while`, `until`, or `if` keyword, and is not a part of an AND or OR list, and is not a pipeline preceded by the ! reserved word, then the shell immediately shall exit.

`−f` The shell shall disable pathname expansion.

`−n` The shell shall read commands but not execute them; this can be used to check for shell script syntax errors. An interactive shell may ignore this option.

`−u` The shell shall write a message to standard error when it tries to expand a variable that is not set and immediately exit. An interactive shell shall not exit.

`−v` The shell shall write its input to standard error as it is read.

`−x` The shell shall write to standard error a trace for each command after it expands the command and before it executes it.

The default for all these options is off (unset) unless the shell was invoked with them on (see `sh` in 4.56). All the positional parameters shall be unset before any new values are assigned.

The remaining arguments shall be assigned in order to the positional parameters. The special parameter # shall be set to reflect the number of positional parameters.

The special argument "--" immediately following the `set` command name can be used to delimit the arguments if the first argument begins with + or −, or to prevent inadvertent listing of all shell variables when there are no arguments. The command `set −−` without *argument* s shall unset all positional parameters and set the special parameter # to zero.

In the obsolescent version, the `set` command name followed by − with no other arguments shall turn off the `−v` and `−x` options without changing the positional parameters. The `set` command name followed by − with other arguments shall turn off the `−v` and `−x` options and assign the arguments to the positional parameters in order.

Exit Status

Zero.

3.14.11.1 `set` Rationale. *(This subclause is not a part of P1003.2)*

The `set −−` form is listed specifically in the Synopsis even though this usage is implied by the utility syntax guidelines. The explanation of this feature removes any ambiguity about whether the `set −−` form might be misinterpreted as being equivalent to `set` without any options or arguments. The functionality of this form has been adopted from the KornShell. In System V, `set −−` only unsets parameters if there is at least one argument; the only way to unset all parameters is to use `shift`. Using the KornShell version should not affect System V scripts because there should be no reason to deliberately issue it without arguments; if it were issued as, say:

```
set −− "$@"
```

and there were in fact no arguments resulting from `$@`, unsetting the parameters would be a no-op anyway.

The `set +` form in earlier drafts was omitted as being an unnecessary duplication of `set` alone and not widespread historical practice.

The noclobber option was changed to `−C` from the `set -o noclobber` option in previous drafts. The `set −o` is used in the KornShell to accept word-length option names, duplicating many of the single-letter names. The noclobber option was changed to a single letter so that the historical `$-` paradigm would not be broken; see 3.5.2.

The following `set` flags were intentionally omitted with the following rationale:

`−h` This flag is related to command name hashing, which is not required for an implementation. It is primarily a performance issue, which is outside the scope of this standard.

–k The –k flag was originally added by Bourne to make it easier for users of prerelease versions of the shell. In early versions of the Bourne shell the construct `set name=value`, had to be used to assign values to shell variables. The problem with –k is that the behavior affects parsing, virtually precluding writing any compilers. To explain the behavior of –k, it is necessary to describe the parsing algorithm, which is implementation defined. For example,

```
set -k; echo name=value
```

and

```
set -k
echo name=value
```

behave differently. The interaction with functions is even more complex. What is more, the –k flag is never needed, since the command line could have been reordered.

–t The –t flag is hard to specify and almost never used. The only known use could be done with here-documents. Moreover, the behavior with `ksh` and `sh` differ. The man page says that it exits after reading and executing one command. What is one command? If the input is `date;date`, `sh` executes both `date` commands, `ksh` does only the first.

Consideration was given to rewriting `set` to simplify its confusing syntax. A specific suggestion was that the `unset` utility should be used to unset options instead of using the non-*getopt*()-able +*option* syntax. However, the conclusion was reached that people were satisfied with the existing practice of using +*option* and there was no compelling reason to modify such widespread existing practice.

Examples :

Write out all variables and their values:

```
set
```

Set $1, $2, and $3 and set $# to 3:

```
set c a b
```

Turn on the –x and –v options:

```
set –xv
```

Unset all positional parameters:

```
set – –
```

Set $1 to the value of x, even if x begins with – or +:

```
set – – "$x"
```

Set the positional parameters to the expansion of x, even if x expands with a leading – or +:

```
set – – $x
```

3.14.12 `shift` — Shift positional parameters

```
shift [n]
```

The positional parameters shall be shifted. Positional parameter 1 shall be assigned the value of parameter $(1+n)$, parameter 2 shall be assigned the value of parameter $(2+n)$, and so forth. The parameters represented by the numbers `$#` down to `$#`$-n+1$ shall be unset, and the parameter # shall be updated to reflect the new number of positional parameters.

The value *n* shall be an unsigned decimal integer less than or equal to the value of the special parameter #. If *n* is not given, it shall be assumed to be `1`. If *n* is `0`, the positional and special parameters shall not be changed.

Exit Status

The exit status shall be >0 if *n*>`$#`; otherwise, it shall be zero.

3.14.12.1 `shift` Rationale. *(This subclause is not a part of P1003.2)*

Example :

```
set a b c d e
shift 2
echo $*
c d e
```

3.14.13 `trap` — Trap signals

```
trap [action condition ...]
```

If *action* is −, the shell shall reset each *condition* to the default value. If *action* is null (`''`), the shell shall ignore each of the specified *condition* s if they arise. Otherwise, the argument *action* shall be read and executed by the shell when one of the corresponding conditions arises. The action of the trap shall override a previous action (either default action or one explicitly set). The value of `$?` after the trap action completes shall be the value it had before the trap was invoked.

The condition can be `EXIT`, `0` (equivalent to `EXIT`), or a signal specified using a symbolic name, without the `SIG` prefix, as listed in Required Signals and Job Control 1 Signals (Table 3-1 and Table 3-2 in POSIX.1 {8}). (For example: `HUP`, `INT`, `QUIT`, `TERM`). Setting a trap for SIGKILL or SIGSTOP produces undefined results.

The environment in which the shell executes a trap on `EXIT` shall be identical to the environment immediately after the last command executed before the trap on `EXIT` was taken.

Each time the trap is invoked, the *action* argument shall be processed in a manner equivalent to:

```
eval "$action"
```

Signals that were ignored on entry to a noninteractive shell cannot be trapped or reset, although no error need be reported when attempting to do so. An interactive shell may reset or catch signals ignored on entry. Traps shall remain in place for a given shell until explicitly changed with another `trap` command.

The `trap` command with no arguments shall write to standard output a list of commands associated with each condition. The format is:

`"trap -- %s %s ...\n"`, *<action>*, *<condition>* ...

The shell shall format the output, including the proper use of quoting, so that it is suitable for re-input to the shell as commands that achieve the same trapping results.

An implementation may allow numeric signal numbers for the conditions as an extension, if and only if the following map of signal numbers to names is true:

Signal Number	Signal Name	Signal Number	Signal Name
1	SIGHUP	9	SIGKILL
2	SIGINT	14	SIGALRM
3	SIGQUIT	15	SIGTERM
6	SIGABRT		

Otherwise, it shall be an error for the application to use numeric signal numbers.

The `trap` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

If the trap name or number is invalid, a nonzero exit status shall be returned; otherwise, zero shall be returned. For both interactive and noninteractive shells, invalid signal names or numbers shall not be considered a syntax error and shall not cause the shell to abort.

3.14.13.1 `trap` Rationale. *(This subclause is not a part of P1003.2)*

Implementations may permit lowercase signal names as an extension. Implementations may also accept the names with the `SIG` prefix; no known historical shell does so. The `trap` and `kill` utilities in POSIX.2 are now consistent in their omission of the `SIG` prefix for signal names. Some `kill` implementations do not allow the prefix and `kill −l` lists the signals without prefixes.

As stated previously, when a subshell is entered, traps are set to the default actions. This does not imply that the `trap` command cannot be used within the subshell to set new traps.

Trapping SIGKILL or SIGSTOP is accepted by some historical implementations, but it does not work. Portable POSIX.2 applications cannot try it.

The output format is not historical practice. Since the output of historical `trap`s is not portable (because numeric signal values are not portable) and had to change to become so, an opportunity was taken to format the output in a way that a shell script could use to save and then later reuse a trap if it wanted. For example:

```
save_traps=$(trap)
. . .
eval "$save_traps"
```

The KornShell uses an `ERR` trap that is triggered whenever `set -e` would cause an exit. This is allowable as an extension, but was not mandated, as other shells have not used it.

The text about the environment for the `EXIT` trap invalidates the behavior of some historical versions of interactive shells which, e.g., close the standard input before executing a trap on 0. For example, in some historical interactive shell sessions the following trap on 0 would always print `--`:

```
trap 'read foo; echo "-$foo-"' 0
```

Examples:

Write out a list of all traps and actions:

```
trap
```

Set a trap so the `logout` utility in the **HOME** directory will execute when the shell terminates:

```
trap '$HOME/logout' EXIT
```
or
```
trap '$HOME/logout' 0
```

Unset traps on INT, QUIT, TERM, and EXIT:

```
trap - INT QUIT TERM EXIT
```

3.14.14 `unset` — Unset values and attributes of variables and functions

`unset [-fv]` *name* ... 1

Each variable or function specified by *name* shall be unset.

If `-v` is specified, *name* refers to a variable name and the shell shall unset it and 1
remove it from the environment. Read-only variables cannot be unset. 1

If `-f` is specified, *name* refers to a function and the shell shall unset the function 1
definition. 1

If neither `-f` nor `-v` is specified, *name* refers to a variable; if a variable by that 1
name does not exist, it is unspecified whether a function by that name, if any, 1
shall be unset. 1

Unsetting a variable or function that was not previously set shall not be considered an error and shall not cause the shell to abort. 1