

## 2. Shell Command Language

This chapter contains the definition of the Shell Command Language.

### 2.1 Shell Introduction

The shell is a command language interpreter. This chapter describes the syntax of that command language as it is used by the [sh](#) utility and the [system\(\)](#) and [popen\(\)](#) functions defined in the System Interfaces volume of POSIX.1-2017.

The shell operates according to the following general overview of operations. The specific details are included in the cited sections of this chapter.

1. The shell reads its input from a file (see [sh](#)), from the **-c** option or from the [system\(\)](#) and [popen\(\)](#) functions defined in the System Interfaces volume of POSIX.1-2017. If the first line of a file of shell commands starts with the characters "#!", the results are unspecified.
2. The shell breaks the input into tokens: words and operators; see [Token Recognition](#).
3. The shell parses the input into simple commands (see [Simple Commands](#)) and compound commands (see [Compound Commands](#)).
4. The shell performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and fields to be treated as a command and arguments; see [wordexp](#).
5. The shell performs redirection (see [Redirection](#)) and removes redirection operators and their operands from the parameter list.
6. The shell executes a function (see [Function Definition Command](#)), built-in (see [Special Built-In Utilities](#)), executable file, or script, giving the names of the arguments as positional parameters numbered 1 to *n*, and the name of the command (or in the case of a function within a script, the name of the script) as the positional parameter numbered 0 (see [Command Search and Execution](#)).
7. The shell optionally waits for the command to complete and collects the exit status (see [Exit Status for Commands](#)).

### 2.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to preserve the literal meaning of the special characters in the next paragraph, prevent reserved words from being recognized as such, and prevent parameter expansion and command substitution within here-document processing (see [Here-Document](#)).

The application shall quote the following characters if they are to represent themselves:

| & ; < > ( ) \$ ` \ " ' <space> <tab> <newline>

and the following may need to be quoted under certain circumstances. That is, these characters may be special depending on conditions described elsewhere in this volume of POSIX.1-2017:

\* ? [ # ~ = %

The various quoting mechanisms are the escape character, single-quotes, and double-quotes. The here-document represents another form of quoting; see [Here-Document](#).

### 2.2.1 Escape Character (Backslash)

A <backslash> that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the <backslash>, the shell shall interpret this as line continuation. The <backslash> and <newline> shall be removed before splitting the input into tokens. Since the escaped <newline> is removed entirely from the input and is not replaced by any white space, it cannot serve as a token separator.

### 2.2.2 Single-Quotes

Enclosing characters in single-quotes ( ' ' ) shall preserve the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

### 2.2.3 Double-Quotes

Enclosing characters in double-quotes ( " " ) shall preserve the literal value of all characters within the double-quotes, with the exception of the characters backquote, <dollar-sign>, and <backslash>, as follows:

\$

The <dollar-sign> shall retain its special meaning introducing parameter expansion (see [Parameter Expansion](#)), a form of command substitution (see [Command Substitution](#)), and arithmetic expansion (see [Arithmetic Expansion](#)).

The input characters within the quoted string that are also enclosed between "\$ ( " and the matching ' ) ' shall not be affected by the double-quotes, but rather shall define that command whose output replaces the "\$ ( . . . ) " when the word is expanded. The tokenizing rules in [Token Recognition](#), not including the alias substitutions in [Alias Substitution](#), shall be applied recursively to find the matching ' ) '.

Within the string of characters from an enclosed "\$ { " to the matching ' } ' , an even number of unescaped double-quotes or single-quotes, if any, shall occur. A preceding <backslash> character shall be used to escape a literal ' { ' or ' } ' . The rule in [Parameter Expansion](#) shall be used to determine the matching ' } ' .

,

The backquote shall retain its special meaning introducing the other form of command substitution (see [Command Substitution](#)). The portion of the quoted string from the initial backquote and the characters up to the next backquote that is not preceded by a <backslash>, having escape characters removed, defines that command whose output replaces " ` . . . ` " when the word is expanded. Either of the following cases produces undefined results:

- A single-quoted or double-quoted string that begins, but does not end, within the " ` . . . ` " sequence
- A " ` . . . ` " sequence that begins, but does not end, within the same double-quoted string

\

The <backslash> shall retain its special meaning as an escape character (see [Escape Character \(Backslash\)](#)) only when followed by one of the following characters when considered special:

\$   `   "   \   <newline>

The application shall ensure that a double-quote is preceded by a <backslash> to be included within double-quotes. The parameter '@' has special meaning inside double-quotes and is described in [Special Parameters](#) .

## 2.3 Token Recognition

The shell shall read its input in terms of lines. (For details about how the shell reads its input, see the description of [sh](#).) The input lines can be of unlimited length. These lines shall be parsed using two major modes: ordinary token recognition and processing of here-documents.

When an **io\_here** token has been recognized by the grammar (see [Shell Grammar](#)), one or more of the subsequent

lines immediately following the next **NEWLINE** token form the body of one or more here-documents and shall be parsed according to the rules of [Here-Document](#).

When it is not processing an **io\_here**, the shell shall break its input into tokens by applying the first applicable rule below to the next character in its input. The token shall be from the current position in the input until a token is delimited according to one of the rules below; the characters forming the token are exactly those in the input, including any quoting characters. If it is indicated that a token is delimited, and no characters have been included in a token, processing shall continue until an actual token is delimited.

1. If the end of input is recognized, the current token (if any) shall be delimited.
2. If the previous character was used as part of an operator and the current character is not quoted and can be used with the previous characters to form an operator, it shall be used as part of that (operator) token.
3. If the previous character was used as part of an operator and the current character cannot be used with the previous characters to form an operator, the operator containing the previous character shall be delimited.
4. If the current character is <backslash>, single-quote, or double-quote and it is not quoted, it shall affect quoting for subsequent characters up to the end of the quoted text. The rules for quoting are as described in [Quoting](#). During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for <newline> joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the <quotation-mark> and the end of the quoted text. The token shall not be delimited by the end of the quoted field.
5. If the current character is an unquoted '\$' or '`', the shell shall identify the start of any candidates for parameter expansion ([Parameter Expansion](#)), command substitution ([Command Substitution](#)), or arithmetic expansion ([Arithmetic Expansion](#)) from their introductory unquoted character sequences: '\$' or "\${", "\$(" or ``, and "\$(", respectively. The shell shall read sufficient input to determine the end of the unit to be expanded (as explained in the cited sections). While processing the characters, if instances of expansions or quoting are found nested within the substitution, the shell shall recursively process them in the manner specified for the construct that is found. The characters found from the beginning of the substitution to its end, allowing for any recursion necessary to recognize embedded constructs, shall be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token shall not be delimited by the end of the substitution.
6. If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) shall be delimited. The current character shall be used as the beginning of the next (operator) token.
7. If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character shall be discarded.
8. If the previous character was part of a word, the current character shall be appended to that word.
9. If the current character is a '#', it and all subsequent characters up to, but excluding, the next <newline> shall be discarded as a comment. The <newline> that ends the line is not considered part of the comment.
10. The current character is used as the start of a new word.

Once a token is delimited, it is categorized as required by the grammar in [Shell Grammar](#).

### 2.3.1 Alias Substitution

After a token has been delimited, but before applying the grammatical rules in [Shell Grammar](#), a resulting word that is identified to be the command name word of a simple command shall be examined to determine whether it is an unquoted, valid alias name. However, reserved words in correct grammatical context shall not be candidates for alias substitution. A valid alias name (see XBD [Alias Name](#)) shall be one that has been defined by the [alias](#) utility and not subsequently undefined using [unalias](#). Implementations also may provide predefined valid aliases that are in effect when the shell is invoked. To prevent infinite loops in recursive aliasing, if the shell is not currently processing an alias of the same name, the word shall be replaced by the value of the alias; otherwise, it shall not be replaced.

If the value of the alias replacing the word ends in a <blank>, the shell shall check the next command word for alias substitution; this process shall continue until a word is found that is not a valid alias or an alias value does not end in a <blank>.

When used as specified by this volume of POSIX.1-2017, alias definitions shall not be inherited by separate invocations of the shell or by the utility execution environments invoked by the shell; see [Shell Execution Environment](#).

## 2.4 Reserved Words

Reserved words are words that have special meaning to the shell; see [Shell Commands](#). The following words shall be recognized as reserved words:

<b>!</b>	<b>do</b>	<b>esac</b>	<b>in</b>
<b>{</b>	<b>done</b>	<b>fi</b>	<b>then</b>
<b>}</b>	<b>elif</b>	<b>for</b>	<b>until</b>
<b>case</b>	<b>else</b>	<b>if</b>	<b>while</b>

This recognition shall only occur when none of the characters is quoted and when the word is used as:

- The first word of a command
- The first word following one of the reserved words other than **case**, **for**, or **in**
- The third word in a **case** command (only **in** is valid in this case)
- The third word in a **for** command (only **in** and **do** are valid in this case)

See the grammar in [Shell Grammar](#).

The following words may be recognized as reserved words on some implementations (when none of the characters are quoted), causing unspecified results:

<b>[[</b>	<b>]]</b>	<b>function</b>	<b>select</b>
-----------	-----------	-----------------	---------------

Words that are the concatenation of a name and a <colon> ( **:** ) are reserved; their use produces unspecified results.

## 2.5 Parameters and Variables

A parameter can be denoted by a name, a number, or one of the special characters listed in [Special Parameters](#). A variable is a parameter denoted by a name.

A parameter is set if it has an assigned value (null is a valid value). Once a variable is set, it can only be unset by using the [unset](#) special built-in command.

### 2.5.1 Positional Parameters

A positional parameter is a parameter denoted by the decimal value represented by one or more digits, other than the single digit 0. The digits denoting the positional parameters shall always be interpreted as a decimal value, even if there is a leading zero. When a positional parameter with more than one digit is specified, the application shall enclose the digits in braces (see [Parameter Expansion](#)). Positional parameters are initially assigned when the shell is invoked (see [sh](#)), temporarily replaced when a shell function is invoked (see [Function Definition Command](#)), and can be reassigned with the [set](#) special built-in command.

### 2.5.2 Special Parameters

Listed below are the special parameters and the values to which they shall expand. Only the values of the special parameters are listed; see [wordexp](#) for a detailed summary of all the stages involved in expanding words.

@

Expands to the positional parameters, starting from one, initially producing one field for each positional parameter that is set. When the expansion occurs in a context where field splitting will be performed, any empty fields may be discarded and each of the non-empty fields shall be further split as described in [Field Splitting](#). When the expansion occurs within double-quotes, the behavior is unspecified unless one of the

following is true:

- Field splitting as described in [Field Splitting](#) would be performed if the expansion were not within double-quotes (regardless of whether field splitting would have any effect; for example, if *IFS* is null).
- The double-quotes are within the *word* of a  $\${parameter:-word}$  or a  $\${parameter:+word}$  expansion (with or without the `<colon>`; see [Parameter Expansion](#)) which would have been subject to field splitting if *parameter* had been expanded instead of *word*.

If one of these conditions is true, the initial fields shall be retained as separate fields, except that if the parameter being expanded was embedded within a word, the first field shall be joined with the beginning part of the original word and the last field shall be joined with the end part of the original word. In all other contexts the results of the expansion are unspecified. If there are no positional parameters, the expansion of '@' shall generate zero fields, even when '@' is within double-quotes; however, if the expansion is embedded within a word which contains one or more other parts that expand to a quoted null string, these null string(s) shall still produce an empty field, except that if the other parts are all within the same double-quotes as the '@', it is unspecified whether the result is zero fields or one empty field.

\*

Expands to the positional parameters, starting from one, initially producing one field for each positional parameter that is set. When the expansion occurs in a context where field splitting will be performed, any empty fields may be discarded and each of the non-empty fields shall be further split as described in [Field Splitting](#). When the expansion occurs in a context where field splitting will not be performed, the initial fields shall be joined to form a single field with the value of each parameter separated by the first character of the *IFS* variable if *IFS* contains at least one character, or separated by a `<space>` if *IFS* is unset, or with no separation if *IFS* is set to a null string.

#

Expands to the decimal number of positional parameters. The command name (parameter 0) shall not be counted in the number given by '#' because it is a special parameter, not a positional parameter.

?

Expands to the decimal exit status of the most recent pipeline (see [Pipelines](#)).

-

(Hyphen.) Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation, by the [set](#) special built-in command, or implicitly by the shell.

\$

Expands to the decimal process ID of the invoked shell. In a subshell (see [Shell Execution Environment](#)), '\$' shall expand to the same value as that of the current shell.

!

Expands to the decimal process ID of the most recent background command (see [Lists](#)) executed from the current shell. (For example, background commands executed from subshells do not affect the value of "\$!" in the current shell environment.) For a pipeline, the process ID is that of the last command in the pipeline.

0

(Zero.) Expands to the name of the shell or shell script. See [sh](#) for a detailed description of how this name is derived.



See the description of the *IFS* variable in [Shell Variables](#).

### 2.5.3 Shell Variables

Variables shall be initialized from the environment (as defined by XBD [Environment Variables](#) and the `exec` function in the System Interfaces volume of POSIX.1-2017) and can be given new values with variable assignment commands. If a variable is initialized from the environment, it shall be marked for export immediately; see the [export](#) special built-in. New variables can be defined and initialized with variable assignments, with the [read](#) or [getopts](#) utilities, with the *name* parameter in a **for** loop, with the  $\${name=word}$  expansion, or with other mechanisms provided as implementation extensions.

The following variables shall affect the execution of the shell:

*ENV*

[UP]  The processing of the *ENV* shell variable shall be supported if the system supports the User Portability Utilities option. 

This variable, when and only when an interactive shell is invoked, shall be subjected to parameter expansion (see [Parameter Expansion](#)) by the shell and the resulting value shall be used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of *ENV* is not an absolute pathname, the results are unspecified. *ENV* shall be ignored if the user's real and effective user IDs or real and effective group IDs are different.

## HOME

The pathname of the user's home directory. The contents of *HOME* are used in tilde expansion (see [Tilde Expansion](#)).

## IFS

A string treated as a list of characters that is used for field splitting, expansion of the ' \* ' special parameter, and to split lines into fields with the [read](#) utility. If the value of *IFS* includes any bytes that do not form part of a valid character, the results of field splitting, expansion of ' \* ', and use of the [read](#) utility are unspecified.

If *IFS* is not set, it shall behave as normal for an unset variable, except that field splitting by the shell and line splitting by the [read](#) utility shall be performed as if the value of *IFS* is <space> <tab> <newline>; see [Field Splitting](#).

The shell shall set *IFS* to <space> <tab> <newline> when it is invoked.

## LANG

Provide a default value for the internationalization variables that are unset or null. (See XBD [Internationalization Variables](#) for the precedence of internationalization variables used to determine the values of locale categories.)

## LC\_ALL

The value of this variable overrides the *LC\_\** variables and *LANG*, as described in XBD [Environment Variables](#).

## LC\_COLLATE

Determine the behavior of range expressions, equivalence classes, and multi-character collating elements within pattern matching.

## LC\_CTYPE

Determine the interpretation of sequences of bytes of text data as characters (for example, single-byte as opposed to multi-byte characters), which characters are defined as letters (character class **alpha**) and <blank> characters (character class **blank**), and the behavior of character classes within pattern matching. Changing the value of *LC\_CTYPE* after the shell has started shall not affect the lexical processing of shell commands in the current shell execution environment or its subshells. Invoking a shell script or performing [exec sh](#) subjects the new shell to the changes in *LC\_CTYPE*.

## LC\_MESSAGES

Determine the language in which messages should be written.

## LINENO

Set by the shell to a decimal number representing the current sequential line number (numbered starting with 1) within a script or function before it executes each command. If the user unsets or resets *LINENO*, the variable may lose its special meaning for the life of the shell. If the shell is not currently executing a script or function, the value of *LINENO* is unspecified. This volume of POSIX.1-2017 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

## NLSPATH

[XSI]  Determine the location of message catalogs for the processing of *LC\_MESSAGES*. 

## PATH

A string formatted as described in XBD [Environment Variables](#), used to effect command interpretation; see [Command Search and Execution](#).

## PPID

Set by the shell to the decimal value of its parent process ID during initialization of the shell. In a subshell (see [Shell Execution Environment](#)), *PPID* shall be set to the same value as that of the parent of the current shell. For example, [echo](#) \$ *PPID* and ([echo](#) \$ *PPID*) would produce the same value.

## PS1

Each time an interactive shell is ready to read a command, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value shall be "\$ ". For users who have specific additional implementation-defined privileges, the default may be another, implementation-defined value. The shell shall replace each instance of the character ' ! ' in *PS1* with the history file number of the next command to be typed. Escaping the ' ! ' with another ' ! ' (that is, " ! ! ") shall place the literal character ' ! ' in the prompt. This volume of POSIX.1-2017 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

## PS2



Each time the user enters a <newline> prior to completing a command line in an interactive shell, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is "> ". This volume of POSIX.1-2017 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

#### PS4

When an execution trace ([set -x](#)) is being performed in an interactive shell, before each line in the execution trace, the value of this variable shall be subjected to parameter expansion and written to standard error. The default value is "+ ". This volume of POSIX.1-2017 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

#### PWD

Set by the shell and by the [cd](#) utility. In the shell the value shall be initialized from the environment as follows. If a value for *PWD* is passed to the shell in the environment when it is executed, the value is an absolute pathname of the current working directory that is no longer than {PATH\_MAX} bytes including the terminating null byte, and the value does not contain any components that are dot or dot-dot, then the shell shall set *PWD* to the value from the environment. Otherwise, if a value for *PWD* is passed to the shell in the environment when it is executed, the value is an absolute pathname of the current working directory, and the value does not contain any components that are dot or dot-dot, then it is unspecified whether the shell sets *PWD* to the value from the environment or sets *PWD* to the pathname that would be output by [pwd -P](#). Otherwise, the [sh](#) utility sets *PWD* to the pathname that would be output by [pwd -P](#). In cases where *PWD* is set to the value from the environment, the value can contain components that refer to files of type symbolic link. In cases where *PWD* is set to the pathname that would be output by [pwd -P](#), if there is insufficient permission on the current working directory, or on any parent of that directory, to determine what that pathname would be, the value of *PWD* is unspecified. Assignments to this variable may be ignored. If an application sets or unsets the value of *PWD*, the behaviors of the [cd](#) and [pwd](#) utilities are unspecified.

## 2.6 Word Expansions

This section describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained in the following sections.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions, and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter '@' within double-quotes, as described in [Special Parameters](#).

The order of word expansion shall be as follows:

1. Tilde expansion (see [Tilde Expansion](#)), parameter expansion (see [Parameter Expansion](#)), command substitution (see [Command Substitution](#)), and arithmetic expansion (see [Arithmetic Expansion](#)) shall be performed, beginning to end. See item 5 in [Token Recognition](#).
2. Field splitting (see [Field Splitting](#)) shall be performed on the portions of the fields generated by step 1, unless *IFS* is null.
3. Pathname expansion (see [Pathname Expansion](#)) shall be performed, unless [set -f](#) is in effect.
4. Quote removal (see [Quote Removal](#)) shall always be performed last.

The expansions described in this section shall occur in the same shell environment as that in which the command is executed.

If the complete expansion appropriate for a word results in an empty field, that empty field shall be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The '\$' character is used to introduce parameter expansion, command substitution, or arithmetic evaluation. If an unquoted '\$' is followed by a character that is not one of the following:

- A numeric character
- The name of one of the special parameters (see [Special Parameters](#))
- A valid first character of a variable name

- A <left-curly-bracket> ( ' { ' )
- A <left-parenthesis>

the result is unspecified.

### 2.6.1 Tilde Expansion

A "tilde-prefix" consists of an unquoted <tilde> character at the beginning of a word, followed by all of the characters preceding the first unquoted <slash> in the word, or all the characters in the word if there is no <slash>. In an assignment (see XBD [Variable Assignment](#)), multiple tilde-prefixes can be used: at the beginning of the word (that is, following the <equals-sign> of the assignment), following any unquoted <colon>, or both. A tilde-prefix in an assignment is terminated by the first unquoted <colon> or <slash>. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the <tilde> are treated as a possible login name from the user database. A portable login name cannot contain characters outside the set given in the description of the *LOGNAME* environment variable in XBD [Other Environment Variables](#). If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix is replaced by the value of the variable *HOME*. If *HOME* is unset, the results are unspecified. Otherwise, the tilde-prefix shall be replaced by a pathname of the initial working directory associated with the login name obtained using the [getpwnam\(\)](#) function as defined in the System Interfaces volume of POSIX.1-2017. If the system does not recognize the login name, the results are undefined.

The pathname resulting from tilde expansion shall be treated as if quoted to prevent it being altered by field splitting and pathname expansion.

### 2.6.2 Parameter Expansion

The format for parameter expansion is as follows:

```
${expression}
```

where *expression* consists of all characters until the matching ' } '. Any ' } ' escaped by a <backslash> or within a quoted string, and characters in embedded arithmetic expansions, command substitutions, and variable expansions, shall not be examined in determining the matching ' } '.

The simplest form for parameter expansion is:

```
${parameter}
```

The value, if any, of *parameter* shall be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when *parameter* is a name and is followed by a character that could be interpreted as part of the name. The matching closing brace shall be determined by counting brace levels, skipping over enclosed quoted strings, and command substitutions.

If the parameter is not enclosed in braces, and is a name, the expansion shall use the longest valid name (see XBD [Name](#)), whether or not the variable represented by that name exists. Otherwise, the parameter is a single-character symbol, and behavior is unspecified if that character is neither a digit nor one of the special parameters (see [Special Parameters](#)).

If a parameter expansion occurs inside double-quotes:

- Pathname expansion shall not be performed on the results of the expansion.
- Field splitting shall not be performed on the results of the expansion.

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of *parameter*, as described below), *word* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *word* is not needed, it shall not be expanded. The ' } ' character that delimits the following parameter expansion modifications shall be determined as



described previously in this section and in [Double-Quotes](#).

**`${parameter:-[word]}`**

**Use Default Values.** If *parameter* is unset or null, the expansion of *word* (or an empty string if *word* is omitted) shall be substituted; otherwise, the value of *parameter* shall be substituted.

**`${parameter:= [word]}`**

**Assign Default Values.** If *parameter* is unset or null, the expansion of *word* (or an empty string if *word* is omitted) shall be assigned to *parameter*. In all cases, the final value of *parameter* shall be substituted. Only variables, not positional parameters or special parameters, can be assigned in this way.

**`${parameter:? [word]}`**

**Indicate Error if Null or Unset.** If *parameter* is unset or null, the expansion of *word* (or a message indicating it is unset if *word* is omitted) shall be written to standard error and the shell exits with a non-zero exit status. Otherwise, the value of *parameter* shall be substituted. An interactive shell need not exit.

**`${parameter:+ [word]}`**

**Use Alternative Value.** If *parameter* is unset or null, null shall be substituted; otherwise, the expansion of *word* (or an empty string if *word* is omitted) shall be substituted.

In the parameter expansions shown previously, use of the <colon> in the format shall result in a test for a parameter that is unset or null; omission of the <colon> shall result in a test for a parameter that is only unset. If *parameter* is '#' and the colon is omitted, the application shall ensure that *word* is specified (this is necessary to avoid ambiguity with the string length expansion). The following table summarizes the effect of the <colon>:

	<i>parameter</i> Set and Not Null	<i>parameter</i> Set But Null	<i>parameter</i> Unset
<b><code>\${parameter:-word}</code></b>	substitute <i>parameter</i>	substitute <i>word</i>	substitute <i>word</i>
<b><code>\${parameter-word}</code></b>	substitute <i>parameter</i>	substitute null	substitute <i>word</i>
<b><code>\${parameter:=word}</code></b>	substitute <i>parameter</i>	assign <i>word</i>	assign <i>word</i>
<b><code>\${parameter=word}</code></b>	substitute <i>parameter</i>	substitute null	assign <i>word</i>
<b><code>\${parameter:?word}</code></b>	substitute <i>parameter</i>	error, exit	error, exit
<b><code>\${parameter?word}</code></b>	substitute <i>parameter</i>	substitute null	error, exit
<b><code>\${parameter:+word}</code></b>	substitute <i>word</i>	substitute null	substitute null
<b><code>\${parameter+word}</code></b>	substitute <i>word</i>	substitute <i>word</i>	substitute null

In all cases shown with "substitute", the expression is replaced with the value shown. In all cases shown with "assign", *parameter* is assigned that value, which also replaces the expression.

**`${#parameter}`**

**String Length.** The length in characters of the value of *parameter* shall be substituted. If *parameter* is '\*' or '@', the result of the expansion is unspecified. If *parameter* is unset and [set -u](#) is in effect, the expansion shall fail.

The following four varieties of parameter expansion provide for substring processing. In each case, pattern matching notation (see [Pattern Matching Notation](#)), rather than regular expression notation, shall be used to evaluate the patterns. If *parameter* is '#', '\*', or '@', the result of the expansion is unspecified. If *parameter* is unset and [set -u](#) is in effect, the expansion shall fail. Enclosing the full parameter expansion string in double-quotes shall not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces shall have this effect. In each variety, if *word* is omitted, the empty pattern shall be used.

**`${parameter% [word]}`**

**Remove Smallest Suffix Pattern.** The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the smallest portion of the suffix matched by the *pattern* deleted. If present, *word* shall not begin with an unquoted '% '.

**`${parameter%% [word]}`**

**Remove Largest Suffix Pattern.** The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the largest portion of the suffix matched by the *pattern* deleted.

**`${parameter# [word]}`**

**Remove Smallest Prefix Pattern.** The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the smallest portion of the prefix matched by the *pattern* deleted. If present, *word* shall not begin with an unquoted '# '.

**`${parameter## [word]}`**

**Remove Largest Prefix Pattern.** The *word* shall be expanded to produce a pattern. The parameter expansion shall then result in *parameter*, with the largest portion of the prefix matched by the *pattern* deleted.

---

The following sections are informative.

## Examples

`${parameter}`

In this example, the effects of omitting braces are demonstrated.

```
a=1
set 2
echo ${a}b-$ab-${1}0-${10}-$10
1b--20--20
```

`${parameter-word}`

This example demonstrates the difference between unset and set to the empty string, as well as the rules for finding the delimiting close brace.

```
foo=asdf
echo ${foo-bar}xyz}
asdfxyz}foo=
echo ${foo-bar}xyz}
xyz}unset foo
echo ${foo-bar}xyz}
barxyz}
```

`${parameter:-word}`

In this example, `ls` is executed only if `x` is null or unset. (The `$(ls)` command substitution notation is explained in [Command Substitution](#).)

```
${x:-$(ls)}
```

`${parameter:=word}`

```
unset X
echo ${X:=abc}
abc
```

`${parameter:?word}`

```
unset posix
echo ${posix:?}
sh: posix: parameter null or not set
```

`${parameter:+word}`

```
set a b c
echo ${3:+posix}
posix
```

`${#parameter}`

```
HOME=/usr/posix
echo ${#HOME}
```

**10**`${parameter%word}`

```
x=file.c
echo ${x%.c}.o
file.o
```

`${parameter%%word}`

```
x=posix/src/std
echo ${x%%/*}
posix
```

`${parameter#word}`

```
x=$HOME/src/cmd
echo ${x#$HOME}
/src/cmd
```

`${parameter##word}`

```
x=/one/two/three
echo ${x##*/}
three
```

The double-quoting of patterns is different depending on where the double-quotes are placed:

`"${x#*}"`

The <asterisk> is a pattern character.

`${x#"*"}`

The literal <asterisk> is quoted and not special.

*End of informative text.*

---

### 2.6.3 Command Substitution

Command substitution allows the output of a command to be substituted in place of the command name itself. Command substitution shall occur when the command is enclosed as follows:

`$( command )`

or (backquoted version):

``command``

The shell shall expand the command substitution by executing *command* in a subshell environment (see [Shell Execution Environment](#)) and replacing the command substitution (the text of *command* plus the enclosing "\$ ( )" or backquotes) with the standard output of the command, removing sequences of one or more <newline> characters at the end of the substitution. Embedded <newline> characters before the end of the output shall not be removed; however, they may be treated as field delimiters and eliminated during field splitting, depending on the value of *IFS* and quoting that is in effect. If the output contains any null bytes, the behavior is unspecified.

Within the backquoted style of command substitution, <backslash> shall retain its literal meaning, except when followed by: '\$', '`', or <backslash>. The search for the matching backquote shall be satisfied by the first unquoted non-escaped backquote; during this search, if a non-escaped backquote is encountered within a shell

comment, a here-document, an embedded command substitution of the  $\$(command)$  form, or a quoted string, undefined results occur. A single-quoted or double-quoted string that begins, but does not end, within the `"`...`"` sequence produces undefined results.

With the  $\$(command)$  form, all characters following the open parenthesis to the matching closing parenthesis constitute the *command*. Any valid shell script can be used for *command*, except a script consisting solely of redirections which produces unspecified results.

The results of command substitution shall not be processed for further tilde expansion, parameter expansion, command substitution, or arithmetic expansion. If a command substitution occurs inside double-quotes, field splitting and pathname expansion shall not be performed on the results of the substitution.

Command substitution can be nested. To specify nesting within the backquoted version, the application shall precede the inner backquotes with `<backslash>` characters; for example:

```
\`command\`
```

The syntax of the shell command language has an ambiguity for expansions beginning with `"$( ( "`, which can introduce an arithmetic expansion or a command substitution that starts with a subshell. Arithmetic expansion has precedence; that is, the shell shall first determine whether it can parse the expansion as an arithmetic expansion and shall only parse the expansion as a command substitution if it determines that it cannot parse the expansion as an arithmetic expansion. The shell need not evaluate nested expansions when performing this determination. If it encounters the end of input without already having determined that it cannot parse the expansion as an arithmetic expansion, the shell shall treat the expansion as an incomplete arithmetic expansion and report a syntax error. A conforming application shall ensure that it separates the `"$( ( "` and `' ( ' "` into two tokens (that is, separate them with white space) in a command substitution that starts with a subshell. For example, a command substitution containing a single subshell could be written as:

```
$( ( command ) )
```

## 2.6.4 Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion shall be as follows:

```
$( ( expression ) )
```

The expression shall be treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell shall expand all tokens in the expression for parameter expansion, command substitution, and quote removal.

Next, the shell shall treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression shall be processed according to the rules given in [Arithmetic Precision and Operations](#), with the following exceptions:

- Only signed long integer arithmetic is required.
- Only the decimal-constant, octal-constant, and hexadecimal-constant constants specified in the ISO C standard, Section 6.4.4.1 are required to be recognized as constants.
- The `sizeof()` operator and the prefix and postfix `"++"` and `"--"` operators are not required.
- Selection, iteration, and jump statements are not supported.

All changes to variables in an arithmetic expression shall be in effect after the arithmetic expansion, as in the parameter expansion `"${x=value}"`.

If the shell variable `x` contains a value that forms a valid integer constant, optionally including a leading `<plus-sign>` or `<hyphen-minus>`, then the arithmetic expansions `"$( ( x ) )"` and `"$( ( $x ) )"` shall return the same value.

As an extension, the shell may recognize arithmetic expressions beyond those listed. The shell may use a signed integer type with a rank larger than the rank of **signed long**. The shell may use a real-floating type instead of **signed long** as long as it does not affect the results in cases where there is no overflow. If the expression is invalid, or the contents of a shell variable used in the expression are not recognized by the shell, the expansion fails and the shell shall write a diagnostic message to standard error indicating the failure.

---

*The following sections are informative.*

## Examples

A simple example using arithmetic expansion:

```
# repeat a command 100 times
x=100
while [ $x -gt 0 ]
do
    command      x=$((x-1))
done
```

*End of informative text.*

---

## 2.6.5 Field Splitting

After parameter expansion ([Parameter Expansion](#)), command substitution ([Command Substitution](#)), and arithmetic expansion ([Arithmetic Expansion](#)), the shell shall scan the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell shall treat each character of the *IFS* as a delimiter and use the delimiters as field terminators to split the results of parameter expansion, command substitution, and arithmetic expansion into fields.

1. If the value of *IFS* is a <space>, <tab>, and <newline>, or if it is unset, any sequence of <space>, <tab>, or <newline> characters at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall delimit a field. For example, the input:

```
<newline><space><tab>foo<tab><tab>bar<space>
```

yields two fields, **foo** and **bar**.

2. If the value of *IFS* is null, no field splitting shall be performed.
3. Otherwise, the following rules shall be applied in sequence. The term "*IFS* white space" is used to mean any sequence (zero or more instances) of white-space characters that are in the *IFS* value (for example, if *IFS* contains <space>/ <comma>/ <tab>, any sequence of <space> and <tab> characters is considered *IFS* white space).
  - a. *IFS* white space shall be ignored at the beginning and end of the input.
  - b. Each occurrence in the input of an *IFS* character that is not *IFS* white space, along with any adjacent *IFS* white space, shall delimit a field, as described previously.
  - c. Non-zero-length *IFS* white space shall delimit a field.

## 2.6.6 Pathname Expansion

After field splitting, if [set -f](#) is not in effect, each field in the resulting command line shall be expanded using the algorithm described in [Pattern Matching Notation](#), qualified by the rules in [Patterns Used for Filename Expansion](#).

### 2.6.7 Quote Removal

The quote characters ( <backslash>, single-quote, and double-quote) that were present in the original word shall be removed unless they have themselves been quoted.

## 2.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see [Shell Execution Environment](#)) or for any command. Redirection operators can be used with numbers representing file descriptors (see XBD [File Descriptor](#)) as described below.

The overall format used for redirection is:

**[*n*] *redir-op word***

The number *n* is an optional decimal number designating the file descriptor number; the application shall ensure it is delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number shall not be recognized as part of the redirection expression. For example:

```
echo \2>a
```

writes the character 2 into file **a**. If any part of *redir-op* is quoted, no redirection expression is recognized. For example:

```
echo 2\>a
```

writes the characters 2>a to standard output. The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

Open files are represented by decimal numbers starting with zero. The largest possible value is implementation-defined; however, all implementations shall support at least 0 to 9, inclusive, for use by the application. These numbers are called "file descriptors". The values 0, 1, and 2 have special meaning and conventional uses and are implied by certain redirection operations; they are referred to as *standard input*, *standard output*, and *standard error*, respectively. Programs usually take their input from standard input, and write output on standard output. Error messages are usually written on standard error. The redirection operators can be preceded by one or more digits (with no intervening <blank> characters allowed) to designate the file descriptor number.

If the redirection operator is "<<" or "<<- ", the word that follows the redirection operator shall be subjected to quote removal; it is unspecified whether any of the other expansions occur. For the other redirection operators, the word that follows the redirection operator shall be subjected to tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion shall not be performed on the word by a non-interactive shell; an interactive shell may perform it, but shall do so only when the expansion would result in one word.

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

A failure to open or create a file shall cause a redirection to fail.

### 2.7.1 Redirecting Input

Input redirection shall cause the file whose name results from the expansion of *word* to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

**[*n*] <*word***

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard input (file descriptor 0).



## 2.7.2 Redirecting Output

The two general formats for redirecting output are:

```
[n]>word
[n]>|word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection shall refer to standard output (file descriptor 1).

Output redirection using the '>' format shall fail if the *noclobber* option is set (see the description of [set -C](#)) and the file named by the expansion of *word* exists and is a regular file. Otherwise, redirection using the '>' or '>|' formats shall cause the file whose name results from the expansion of *word* to be created and opened for output on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

## 2.7.3 Appending Redirected Output

Appended output redirection shall cause the file whose name results from the expansion of *word* to be opened for output on the designated file descriptor. The file is opened as if the [open\(\)](#) function as defined in the System Interfaces volume of POSIX.1-2017 was called with the `O_APPEND` flag. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

```
[n]>>word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection refers to standard output (file descriptor 1).

## 2.7.4 Here-Document

The redirection operators "<<" and "<<-" both allow redirection of subsequent lines read by the shell to the input of a command. The redirected lines are known as a "here-document".

The here-document shall be treated as a single word that begins after the next <newline> and continues until there is a line containing only the delimiter and a <newline>, with no <blank> characters in between. Then the next here-document starts, if there is one. The format is as follows:

```
[n]<<word
    here-document
delimiter
```

where the optional *n* represents the file descriptor number. If the number is omitted, the here-document refers to standard input (file descriptor 0). It is unspecified whether the file descriptor is opened as a regular file, a special file, or a pipe. Portable applications cannot rely on the file descriptor being seekable (see XSH [lseek](#)).

If any part of *word* is quoted, the delimiter shall be formed by performing quote removal on *word*, and the here-document lines shall not be expanded. Otherwise, the delimiter shall be the *word* itself.

If no part of *word* is quoted, all lines of the here-document shall be expanded for parameter expansion, command substitution, and arithmetic expansion. In this case, the <backslash> in the input behaves as the <backslash> inside double-quotes (see [Double-Quotes](#)). However, the double-quote character ( ' ' ) shall not be treated specially within a here-document, except when the double-quote appears within "\$ ( ) ", "` ` ", or "\${ } ".

If the redirection operator is "<<-", all leading <tab> characters shall be stripped from input lines and the line containing the trailing delimiter. If more than one "<<" or "<<-" operator is specified on a line, the here-document associated with the first operator shall be supplied first by the application and shall be read first by the shell.

When a here-document is read from a terminal device and the shell is interactive, it shall write the contents of the variable *PS2*, processed as described in [Shell Variables](#), to standard error before reading each line of input until the

delimiter has been recognized.

---

*The following sections are informative.*

## Examples

An example of a here-document follows:

```
cat <<eof1; cat <<eof2
Hi,
eof1
Helene.
eof2
```

*End of informative text.*

---

### 2.7.5 Duplicating an Input File Descriptor

The redirection operator:

**[*n*]** <&*word*

shall duplicate one input file descriptor from another, or shall close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard input if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for input, a redirection error shall result; see [Consequences of Shell Errors](#). If *word* evaluates to ' - ', file descriptor *n*, or standard input if *n* is not specified, shall be closed. Attempts to close a file descriptor that is not open shall not constitute an error. If *word* evaluates to something else, the behavior is unspecified.

### 2.7.6 Duplicating an Output File Descriptor

The redirection operator:

**[*n*]** >&*word*

shall duplicate one output file descriptor from another, or shall close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard output if *n* is not specified, shall be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for output, a redirection error shall result; see [Consequences of Shell Errors](#). If *word* evaluates to ' - ', file descriptor *n*, or standard output if *n* is not specified, is closed. Attempts to close a file descriptor that is not open shall not constitute an error. If *word* evaluates to something else, the behavior is unspecified.

### 2.7.7 Open File Descriptors for Reading and Writing

The redirection operator:

**[*n*]** <>*word*

shall cause the file whose name is the expansion of *word* to be opened for both reading and writing on the file descriptor denoted by *n*, or standard input if *n* is not specified. If the file does not exist, it shall be created.

## 2.8 Exit Status and Errors

### 2.8.1 Consequences of Shell Errors

Certain errors shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

Error	Non-Interactive Shell	Interactive Shell	Shell Diagnostic Message Required
Shell language syntax error	shall exit	shall not exit	yes
Special built-in utility error	shall exit	shall not exit	no <sup>1</sup>
Other utility (not a special built-in) error	shall not exit	shall not exit	no <sup>2</sup>
Redirection error with special built-in utilities	shall exit	shall not exit	yes
Redirection error with compound commands	may exit <sup>3</sup>	shall not exit	yes
Redirection error with function execution	may exit <sup>3</sup>	shall not exit	yes
Redirection error with other utilities (not special built-ins)	shall not exit	shall not exit	yes
Variable assignment error	shall exit	shall not exit	yes
Expansion error	shall exit	shall not exit	yes
Command not found	may exit	shall not exit	yes

Notes:

1. Although special built-ins are part of the shell, a diagnostic message written by a special built-in is not considered to be a shell diagnostic message, and can be redirected like any other utility.
2. The shell is not required to write a diagnostic message, but the utility itself shall write a diagnostic message if required to do so.
3. A future version of this standard may require the shell to not exit in this condition.

An expansion error is one that occurs when the shell expansions define in [wordexp](#) are carried out (for example, "\$ {x!y} ", because ' ! ' is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as "shall exit" or "may exit" occur in a subshell environment, the shell shall (respectively, may) exit from the subshell environment with a non-zero status and continue in the environment from which that subshell environment was invoked.

In all of the cases shown in the table where an interactive shell is required not to exit, the shell shall not perform any further processing of the command in which the error occurred.

## 2.8.2 Exit Status for Commands

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities is documented in this section. The exit status of the standard utilities is documented in their respective sections.

If a command is not found, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status shall be between 1 and 125 inclusive.

Internally, for purposes of deciding whether a command exits with a non-zero exit status, the shell shall recognize the entire status value retrieved for the command by the equivalent of the [wait\(\)](#) function WEXITSTATUS macro (as defined in the System Interfaces volume of POSIX.1-2017). When reporting the exit status with the special parameter ' ? ', the shell shall report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal shall be reported as greater than 128.

## 2.9 Shell Commands

This section describes the basic structure of shell commands. The following command descriptions each describe a format of the command that is only used to aid the reader in recognizing the command type, and does not formally represent the syntax. In particular, the representations include spacing between tokens in some places where <blank>s would not be necessary (when one of the tokens is an operator). Each description discusses the semantics of the command; for a formal definition of the command language, consult [Shell Grammar](#).

A *command* is one of the following:

- Simple command (see [Simple Commands](#))
- Pipeline (see [Pipelines](#))
- List compound-list (see [Lists](#))
- Compound command (see [Compound Commands](#))
- Function definition (see [Function Definition Command](#))

Unless otherwise stated, the exit status of a command shall be that of the last simple command executed by the command. There shall be no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG\_MAX}, and so on).

### 2.9.1 Simple Commands

A "simple command" is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (that is, when any conditional construct such as an AND-OR list or a **case** statement has not bypassed the simple command), the following expansions, assignments, and redirections shall all be performed from the beginning of the command text to the end:

1. The words that are recognized as variable assignments or redirections according to [Shell Grammar Rules](#) are saved for processing in steps 3 and 4.
2. The words that are not variable assignments or redirections shall be expanded. If any fields remain following their expansion, the first field shall be considered the command name and remaining fields are the arguments for the command.
3. Redirections shall be performed as described in [Redirection](#).
4. Each variable assignment shall be expanded for tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal prior to assigning the value.

In the preceding list, the order of steps 3 and 4 may be reversed if no command name results from step 2 or if the command name matches the name of a special built-in utility; see [Special Built-In Utilities](#).

Variable assignments shall be performed as follows:

- If no command name results, variable assignments shall affect the current execution environment.
- If the command name is not a special built-in utility or function, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment except as a side-effect of the expansions performed in step 4. In this case it is unspecified:
  - Whether or not the assignments are visible for subsequent expansions in step 4
  - Whether variable assignments made as side-effects of these expansions are visible for subsequent expansions in step 4, or in the current shell execution environment, or both
- If the command name is a standard utility implemented as a function (see XBD [Utility](#)), the effect of variable assignments shall be as if the utility was not implemented as a function.
- If the command name is a special built-in utility, variable assignments shall affect the current execution environment. Unless the [set](#) **-a** option is on (see [set](#)), it is unspecified:

- Whether or not the variables gain the *export* attribute during the execution of the special built-in utility
- Whether or not *export* attributes gained as a result of the variable assignments persist after the completion of the special built-in utility
- If the command name is a function that is not a standard utility implemented as a function, variable assignments shall affect the current execution environment during the execution of the function. It is unspecified:
  - Whether or not the variable assignments persist after the completion of the function
  - Whether or not the variables gain the *export* attribute during the execution of the function
  - Whether or not *export* attributes gained as a result of the variable assignments persist after the completion of the function (if variable assignments persist after the completion of the function)

If any of the variable assignments attempt to assign a value to a variable for which the *readonly* attribute is set in the current shell environment (regardless of whether the assignment is made in that environment), a variable assignment error shall occur. See [Consequences of Shell Errors](#) for the consequences of these errors.

If there is no command name, any redirections shall be performed in a subshell environment; it is unspecified whether this subshell environment is the same one as that used for a command substitution within the command. (To affect the current execution environment, see the [exec](#) special built-in.) If any of the redirections performed in the current shell execution environment fail, the command shall immediately fail with an exit status greater than zero, and the shell shall write an error message indicating the failure. See [Consequences of Shell Errors](#) for the consequences of these failures on interactive and non-interactive shells.

If there is a command name, execution shall continue as described in [Command Search and Execution](#). If there is no command name, but the command contained a command substitution, the command shall complete with the exit status of the last command substitution performed. Otherwise, the command shall complete with a zero exit status.

## Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions shall be performed:

1. If the command name does not contain any <slash> characters, the first successful step in the following sequence shall occur:
  - a. If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.
  - b. If the command name matches the name of a utility listed in the following table, the results are unspecified.

<i>alloc</i>	<i>comparguments</i>	<i>comptr</i>	<i>history</i>	<i>pushd</i>
<i>autoload</i>	<i>compcall</i>	<i>compvalues</i>	<i>hist</i>	<i>readarray</i>
<i>bind</i>	<i>compctl</i>	<i>declare</i>	<i>let</i>	<i>repeat</i>
<i>bindkey</i>	<i>compdescribe</i>	<i>dirs</i>	<i>local</i>	<i>savehistory</i>
<i>builtin</i>	<i>compfiles</i>	<i>disable</i>	<i>login</i>	<i>source</i>
<i>bye</i>	<i>compngen</i>	<i>disown</i>	<i>logout</i>	<i>shopt</i>
<i>caller</i>	<i>compgroups</i>	<i>dosh</i>	<i>map</i>	<i>stop</i>
<i>cap</i>	<i>complete</i>	<i>echotc</i>	<i>mapfile</i>	<i>suspend</i>
<i>chdir</i>	<i>compquote</i>	<i>echo</i>	<i>popd</i>	<i>typeset</i>
<i>clone</i>	<i>comptags</i>	<i>help</i>	<i>print</i>	<i>whence</i>

- c. If the command name matches the name of a function known to this shell, the function shall be invoked as described in [Function Definition Command](#). If the implementation has provided a standard utility in the form of a function, it shall not be recognized at this point. It shall be invoked in conjunction with the path search in step 1e.
  - d. If the command name matches the name `[XSI]` of the *type* or *ulimit* utility, or `[X]` of a utility listed in

the following table, that utility shall be invoked.

<a href="#"><i>alias</i></a>	<a href="#"><i>false</i></a>	<a href="#"><i>hash</i></a>	<a href="#"><i>pwd</i></a>	<a href="#"><i>unalias</i></a>
<a href="#"><i>bg</i></a>	<a href="#"><i>fc</i></a>	<a href="#"><i>jobs</i></a>	<a href="#"><i>read</i></a>	<a href="#"><i>wait</i></a>
<a href="#"><i>cd</i></a>	<a href="#"><i>fg</i></a>	<a href="#"><i>kill</i></a>	<a href="#"><i>true</i></a>	
<a href="#"><i>command</i></a>	<a href="#"><i>getopts</i></a>	<a href="#"><i>newgrp</i></a>	<a href="#"><i>umask</i></a>	

- e. Otherwise, the command shall be searched for using the *PATH* environment variable as described in XBD [Environment Variables](#) :

- i. If the search is successful:

- a. If the system has implemented the utility as a regular built-in or as a shell function, it shall be invoked at this point in the path search.
- b. Otherwise, the shell executes the utility in a separate utility environment (see [Shell Execution Environment](#)) with actions equivalent to calling the [exec\(\)](#) function as defined in the System Interfaces volume of POSIX.1-2017 with the *path* argument set to the pathname resulting from the search, *arg0* set to the command name, and the remaining [exec\(\)](#) arguments set to the command arguments (if any) and the null terminator.

If the [exec\(\)](#) function fails due to an error equivalent to the [ENOEXEC] error defined in the System Interfaces volume of POSIX.1-2017, the shell shall execute a command equivalent to having a shell invoked with the pathname resulting from the search as its first operand, with any remaining arguments passed to the new shell, except that the value of "\$0" in the new shell may be set to the command name. If the executable file is not a text file, the shell may bypass this command execution. In this case, it shall write an error message, and shall return an exit status of 126.

It is unspecified whether environment variables that were passed to the shell when it was invoked, but were not used to initialize shell variables (see [Shell Variables](#)) because they had invalid names, are included in the environment passed to [exec\(\)](#) and (if [exec\(\)](#) fails as described above) to the new shell.

Once a utility has been searched for and found (either as a result of this specific search or as part of an unspecified shell start-up activity), an implementation may remember its location and need not search for the utility again unless the *PATH* variable has been the subject of an assignment. If the remembered location fails for a subsequent invocation, the shell shall repeat the search to find the new location for the utility, if any.

- ii. If the search is unsuccessful, the command shall fail with an exit status of 127 and the shell shall write an error message.

2. If the command name contains at least one <slash>, the shell shall execute the utility in a separate utility environment with actions equivalent to calling the [exec\(\)](#) function defined in the System Interfaces volume of POSIX.1-2017 with the *path* and *arg0* arguments set to the command name, and the remaining [exec\(\)](#) arguments set to the command arguments (if any) and the null terminator.

If the [exec\(\)](#) function fails due to an error equivalent to the [ENOEXEC] error, the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, with any remaining arguments passed to the new shell. If the executable file is not a text file, the shell may bypass this command execution. In this case, it shall write an error message and shall return an exit status of 126.

It is unspecified whether environment variables that were passed to the shell when it was invoked, but were not used to initialize shell variables (see [Shell Variables](#)) because they had invalid names, are included in the environment passed to [exec\(\)](#) and (if [exec\(\)](#) fails as described above) to the new shell.

If the utility would be executed with file descriptor 0, 1, or 2 closed, implementations may execute the utility with the file descriptor open to an unspecified file. If a standard utility or a conforming application is executed with file descriptor 0 not open for reading or with file descriptor 1 or 2 not open for writing, the environment in which the utility or application is executed shall be deemed non-conforming, and consequently the utility or application might not behave as described in this standard.



## 2.9.2 Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator ' | '. For each command but the last, the shell shall connect the standard output of the command to the standard input of the next command as if by creating a pipe and passing the write end of the pipe as the standard output of the command and the read end of the pipe as the standard input of the next command.

The format for a pipeline is:

```
[!] command1 [ | command2 ... ]
```

If the pipeline begins with the reserved word ! and *command1* is a subshell command, the application shall ensure that the ( operator at the beginning of *command1* is separated from the ! by one or more <blank> characters. The behavior of the reserved word ! immediately followed by the ( operator is unspecified.

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see [Redirection](#)).

If the pipeline is not in the background (see [Asynchronous Lists](#)), the shell shall wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

## Exit Status

If the pipeline does not begin with the ! reserved word, the exit status shall be the exit status of the last command specified in the pipeline. Otherwise, the exit status shall be the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status shall be 1; if the last command returns greater than zero, the exit status shall be zero.

## 2.9.3 Lists

An *AND-OR list* is a sequence of one or more pipelines separated by the operators "&&" and " | " .

A *list* is a sequence of one or more AND-OR lists separated by the operators ';' and '&' .

The operators "&&" and " | " shall have equal precedence and shall be evaluated with left associativity. For example, both of the following commands write solely **bar** to standard output:

```
false && echo foo || echo bar
true || echo foo && echo bar
```

A ';' separator or a ';' or <newline> terminator shall cause the preceding AND-OR list to be executed sequentially; an '&' separator or terminator shall cause asynchronous execution of the preceding AND-OR list.

The term "compound-list" is derived from the grammar in [Shell Grammar](#); it is equivalent to a sequence of *lists*, separated by <newline> characters, that can be preceded or followed by an arbitrary number of <newline> characters.

---

*The following sections are informative.*

## Examples

The following is an example that illustrates <newline> characters in compound-lists:

```
while
    # a couple of <newline>s
```

```
# a list
date && who || ls; cat file
# a couple of <newline>s

# another list
wc file > output & true

do
    # 2 lists
    ls
    cat file
done
```

*End of informative text.*

---

## Asynchronous Lists

If a command is terminated by the control operator `&` ( `'&'` ), the shell shall execute the command asynchronously in a subshell. This means that the shell shall not wait for the command to finish before executing the next command.

The format for running a command in the background is:

```
command1 & [command2 & ... ]
```

If job control is disabled (see [set](#), `-m`), the standard input for an asynchronous list, before any explicit redirections are performed, shall be considered to be assigned to a file that has the same properties as `/dev/null`. This shall not happen if job control is enabled. In all cases, explicit redirection of standard input shall override this activity.

When an element of an asynchronous list (the portion of the list ended by an `&`, such as *command1*, above) is started by the shell, the process ID of the last command in the asynchronous list element shall become known in the current shell execution environment; see [Shell Execution Environment](#). This process ID shall remain known until:

1. The command terminates and the application waits for the process ID.
2. Another asynchronous list is invoked before `"$!"` (corresponding to the previous asynchronous list) is expanded in the current execution environment.

The implementation need not retain more than the `{CHILD_MAX}` most recent entries in its list of known process IDs in the current shell execution environment.

## Exit Status

The exit status of an asynchronous list shall be zero.

## Sequential Lists

Commands that are separated by a `<semicolon>` ( `' ; '` ) shall be executed sequentially.

The format for executing commands sequentially shall be:

```
command1 [ ; command2 ] . . .
```

Each command shall be expanded and executed in the order specified.

## Exit Status

The exit status of a sequential list shall be the exit status of the last command in the list.

## AND Lists

The control operator "&&" denotes an AND list. The format shall be:

```
command1 [ && command2 ] . . .
```

First *command1* shall be executed. If its exit status is zero, *command2* shall be executed, and so on, until a command has a non-zero exit status or there are no more commands left to execute. The commands are expanded only if they are executed.

## Exit Status

The exit status of an AND list shall be the exit status of the last command that is executed in the list.

## OR Lists

The control operator "|" denotes an OR List. The format shall be:

```
command1 [ | command2 ] . . .
```

First, *command1* shall be executed. If its exit status is non-zero, *command2* shall be executed, and so on, until a command has a zero exit status or there are no more commands left to execute.

## Exit Status

The exit status of an OR list shall be the exit status of the last command that is executed in the list.

## 2.9.4 Compound Commands

The shell has several programming constructs that are "compound commands", which provide control flow for commands. Each of these compound commands has a reserved word or control operator at the beginning, and a corresponding terminator reserved word or operator at the end. In addition, each can be followed by redirections on the same line as the terminator. Each redirection shall apply to all the commands within the compound command that do not explicitly override that redirection.

## Grouping Commands

The format for grouping commands is as follows:

```
( compound-list )
```

Execute *compound-list* in a subshell environment; see [Shell Execution Environment](#). Variable assignments and built-in commands that affect the environment shall not remain in effect after the list finishes.

If a character sequence beginning with "(" would be parsed by the shell as an arithmetic expansion if

preceded by a '\$', shells which implement an extension whereby "( (*expression*) )" is evaluated as an arithmetic expression may treat the "( (" as introducing as an arithmetic evaluation instead of a grouping command. A conforming application shall ensure that it separates the two leading ' (' characters with white space to prevent the shell from performing an arithmetic evaluation.

```
{ compound-list ; }
```

Execute *compound-list* in the current process environment. The semicolon shown here is an example of a control operator delimiting the } reserved word. Other delimiters are possible, as shown in [Shell Grammar](#); a <newline> is frequently used.

## Exit Status

The exit status of a grouping command shall be the exit status of *compound-list*.

## The for Loop

The **for** loop shall execute a sequence of commands for each member in a list of *items*. The **for** loop requires that the reserved words **do** and **done** be used to delimit the sequence of commands.

The format for the **for** loop is as follows:

```
for name [ in [word ... ] ]
do
    compound-list
done
```

First, the list of words following **in** shall be expanded to generate a list of items. Then, the variable *name* shall be set to each item, in turn, and the *compound-list* executed each time. If no items result from the expansion, the *compound-list* shall not be executed. Omitting:

```
in word...
```

shall be equivalent to:

```
in "$@"
```

## Exit Status

The exit status of a **for** command shall be the exit status of the last command that executes. If there are no items, the exit status shall be zero.

## Case Conditional Construct

The conditional construct **case** shall execute the *compound-list* corresponding to the first one of several *patterns* (see [Pattern Matching Notation](#)) that is matched by the string resulting from the tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal of the given word. The reserved word **in** shall denote the beginning of the patterns to be matched. Multiple patterns with the same *compound-list* shall be delimited by the '|' symbol. The control operator ')' terminates a list of patterns corresponding to a given action. The *compound-list* for each list of patterns, with the possible exception of the last, shall be terminated with ";;". The **case** construct terminates with the reserved word **esac** (**case** reversed).

The format for the **case** construct is as follows:

```
case word in
    [(] pattern1 ) compound-list ;;
```

```

    [[ ( pattern [ | pattern ] ... ) compound-list ;; ] ...
    [[ ( pattern [ | pattern ] ... ) compound-list ]
esac

```

The " ; ; " is optional for the last *compound-list*.

In order from the beginning to the end of the **case** statement, each *pattern* that labels a *compound-list* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion, and the result of these expansions shall be compared against the expansion of *word*, according to the rules described in [Pattern Matching Notation](#) (which also describes the effect of quoting parts of the pattern). After the first match, no more patterns shall be expanded, and the *compound-list* shall be executed. The order of expansion and comparison of multiple *patterns* that label a *compound-list* statement is unspecified.

## Exit Status

The exit status of **case** shall be zero if no patterns are matched. Otherwise, the exit status shall be the exit status of the last command executed in the *compound-list*.

## The if Conditional Construct

The **if** command shall execute a *compound-list* and use its exit status to determine whether to execute another *compound-list*.

The format for the **if** construct is as follows:

```

if compound-list
then
    compound-list
[elif compound-list
then
    compound-list] ...
[else
    compound-list]
fi

```

The **if** *compound-list* shall be executed; if its exit status is zero, the **then** *compound-list* shall be executed and the command shall complete. Otherwise, each **elif** *compound-list* shall be executed, in turn, and if its exit status is zero, the **then** *compound-list* shall be executed and the command shall complete. Otherwise, the **else** *compound-list* shall be executed.

## Exit Status

The exit status of the **if** command shall be the exit status of the **then** or **else** *compound-list* that was executed, or zero, if none was executed.

## The while Loop

The **while** loop shall continuously execute one *compound-list* as long as another *compound-list* has a zero exit status.

The format of the **while** loop is as follows:

```

while compound-list-1

```

```
do
    compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a non-zero exit status, the **while** command shall complete. Otherwise, the *compound-list-2* shall be executed, and the process shall repeat.

### Exit Status

The exit status of the **while** loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

### The until Loop

The **until** loop shall continuously execute one *compound-list* as long as another *compound-list* has a non-zero exit status.

The format of the **until** loop is as follows:

```
until compound-list-1
do
    compound-list-2
done
```

The *compound-list-1* shall be executed, and if it has a zero exit status, the **until** command completes. Otherwise, the *compound-list-2* shall be executed, and the process repeats.

### Exit Status

The exit status of the **until** loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

## 2.9.5 Function Definition Command

A function is a user-defined name that is used as a simple command to call a compound command with new positional parameters. A function is defined with a "function definition command".

The format of a function definition command is as follows:

```
fname ( ) compound-command [io-redirect ...]
```

The function is named *fname*; the application shall ensure that it is a name (see XBD [Name](#)) and that it is not the name of a special built-in utility. An implementation may allow other characters in a function name as an extension. The implementation shall maintain separate name spaces for functions and variables.

The argument *compound-command* represents a compound command, as described in [Compound Commands](#).

When the function is declared, none of the expansions in [wordexp](#) shall be performed on the text in *compound-command* or *io-redirect*; all expansions shall be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* shall be performed during the execution of the function itself, not the function definition. See [Consequences of Shell Errors](#) for the consequences of failures of these operations on interactive and non-interactive shells.

When a function is executed, it shall have the syntax-error properties described for special built-in utilities in the first item in the enumerated list at the beginning of [Special Built-In Utilities](#).



The *compound-command* shall be executed whenever the function name is specified as the name of a simple command (see [Command Search and Execution](#)). The operands to the command temporarily shall become the positional parameters during the execution of the *compound-command*; the special parameter '#' also shall be changed to reflect the number of operands. The special parameter 0 shall be unchanged. When the function completes, the values of the positional parameters and the special parameter '#' shall be restored to the values they had before the function was executed. If the special built-in [return](#) (see [return](#)) is executed in the *compound-command*, the function completes and execution shall resume with the next command after the function call.

## Exit Status

The exit status of a function definition shall be zero if the function was declared successfully; otherwise, it shall be greater than zero. The exit status of a function invocation shall be the exit status of the last command executed by the function.

## 2.10. Shell Grammar

The following grammar defines the Shell Command Language. This formal syntax shall take precedence over the preceding text syntax description.

### 2.10.1 Shell Grammar Lexical Conventions

The input language to the shell must be first recognized at the character level. The resulting tokens shall be classified by their immediate context according to the following rules (applied in order). These rules shall be used to determine what a "token" is that is subject to parsing at the token level. The rules for token recognition in [Token Recognition](#) shall apply.

1. If the token is an operator, the token identifier for that operator shall result.
2. If the string consists solely of digits and the delimiter character is one of '<' or '>', the token identifier **IO\_NUMBER** shall be returned.
3. Otherwise, the token identifier **TOKEN** results.

Further distinction on **TOKEN** is context-dependent. It may be that the same **TOKEN** yields **WORD**, a **NAME**, an **ASSIGNMENT\_WORD**, or one of the reserved words below, dependent upon the context. Some of the productions in the grammar below are annotated with a rule number from the following list. When a **TOKEN** is seen where one of those annotated productions could be used to reduce the symbol, the applicable rule shall be applied to convert the token identifier type of the **TOKEN** to a token identifier acceptable at that point in the grammar. The reduction shall then proceed based upon the token identifier type yielded by the rule applied. When more than one rule applies, the highest numbered rule shall apply (which in turn may refer to another rule). (Note that except in rule 7, the presence of an '=' in the token has no effect.)

The **WORD** tokens shall have the word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

### 2.10.2 Shell Grammar Rules

1. [Command Name]

When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, the token **WORD** shall be returned. Also, if the parser is in any state where only a reserved word could be the next correct token, proceed as above.

#### Note:

Because at this point <quotation-mark> characters are retained in the token, quoted strings cannot be recognized as reserved words. This rule also implies that reserved words are not recognized except in certain positions in the input, such as after a <newline> or <semicolon>; the grammar presumes that if the reserved word is intended, it is properly delimited by the user, and does not attempt to reflect that requirement directly. Also note that line joining is done before tokenization, as described in [Escape Character \(Backslash\)](#), so escaped <newline> characters are already removed at this point.

Rule 1 is not directly referenced in the grammar, but is referred to by other rules, or applies globally.

2. [Redirection to or from filename]

The expansions specified in [Redirection](#) shall occur. As specified there, exactly one field can result (or the result is unspecified), and there are additional requirements on pathname expansion.

3. [Redirection from here-document]

Quote removal shall be applied to the word to determine the delimiter that is used to find the end of the here-document that begins after the next <newline>.

4. [Case statement termination]

When the **TOKEN** is exactly the reserved word **esac**, the token identifier for **esac** shall result. Otherwise, the token **WORD** shall be returned.

5. [ **NAME** in **for**]

When the **TOKEN** meets the requirements for a name (see XBD [Name](#)), the token identifier **NAME** shall result. Otherwise, the token **WORD** shall be returned.

6. [Third word of **for** and **case**]

a. [ **case** only]

When the **TOKEN** is exactly the reserved word **in**, the token identifier for **in** shall result. Otherwise, the token **WORD** shall be returned.

b. [ **for** only]

When the **TOKEN** is exactly the reserved word **in** or **do**, the token identifier for **in** or **do** shall result, respectively. Otherwise, the token **WORD** shall be returned.

(For a. and b.: As indicated in the grammar, a *linebreak* precedes the tokens **in** and **do**. If <newline> characters are present at the indicated location, it is the token after them that is treated in this fashion.)

7. [Assignment preceding command name]

a. [When the first word]

If the **TOKEN** does not contain the character '=' , rule 1 is applied. Otherwise, 7b shall be applied.

b. [Not the first word]

If the **TOKEN** contains an unquoted (as determined while applying rule 4 from [Token Recognition](#)) <equals-sign> character that is not part of an embedded parameter expansion, command substitution, or arithmetic expansion construct (as determined while applying rule 5 from [Token Recognition](#)):

- If the **TOKEN** begins with '=' , then rule 1 shall be applied.
- If all the characters in the **TOKEN** preceding the first such <equals-sign> form a valid name (see XBD [Name](#)), the token **ASSIGNMENT\_WORD** shall be returned.
- Otherwise, it is unspecified whether rule 1 is applied or **ASSIGNMENT\_WORD** is returned.

Otherwise, rule 1 shall be applied.

Assignment to the name within a returned **ASSIGNMENT\_WORD** token shall occur as specified in [Simple Commands](#).

8. [ **NAME** in function]

When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, when the **TOKEN** meets the requirements for a name, the token identifier **NAME** shall result. Otherwise, rule 7 applies.

9. [Body of function]

Word expansion and assignment shall never occur, even when required by the rules above, when this rule is being parsed. Each **TOKEN** that might either be expanded or have assignment applied to it shall instead be returned as a single **WORD** consisting only of characters that are exactly the token described in [Token Recognition](#).

```

/* -----
   The grammar symbols
   ----- */

%token  WORD
%token  ASSIGNMENT_WORD
%token  NAME
%token  NEWLINE
%token  IO_NUMBER


/* The following are the operators (see XBD Operator)
   containing more than one character. */


%token  AND_IF      OR_IF      DSEMI
/*      '&&'        '||'       ';;'      */

%token  DLESS      DGREAT     LESSAND    GREATAND    LESSGREAT    DLESSDASH
/*      '<<'        '>>'       '<&'       '>&'        '<>'         '<<- '      */

%token  CLOBBER
/*      '>|'       */

/* The following are the reserved words. */


%token  If      Then      Else      Elif      Fi      Do      Done
/*      'if'     'then'    'else'   'elif'   'fi'     'do'     'done'   */

%token  Case      Esac      While      Until      For
/*      'case'    'esac'    'while'  'until'   'for'     */

/* These are reserved words, not operator tokens, and are
   recognized when reserved words are recognized. */

%token  Lbrace      Rbrace      Bang

```

```
/*      '{'      '}'      '!'      */
```

```
%token  In
```

```
/*      'in'      */
```

```
/* -----
   The Grammar
   ----- */
```

```
%start program
```

```
%%
```

```
program      : linebreak complete_commands linebreak
              | linebreak
```

```
;
```

```
complete_commands: complete_commands newline_list complete_command
                  |                               complete_command
```

```
;
```

```
complete_command : list separator_op
                  | list
```

```
;
```

```
list          : list separator_op and_or
                  |                               and_or
```

```
;
```

```
and_or        :                               pipeline
                  | and_or AND_IF linebreak pipeline
                  | and_or OR_IF  linebreak pipeline
```

```
;
```

```
pipeline      :       pipe_sequence
                  | Bang pipe_sequence
```

```
;
```

```
pipe_sequence :                               command
                  | pipe_sequence '|' linebreak command
```

```
;
```

```
command       : simple_command
                  | compound_command
                  | compound_command redirect_list
                  | function_definition
```

```
;
```

```
compound_command : brace_group
                  | subshell
                  | for_clause
                  | case_clause
                  | if_clause
                  | while_clause
                  | until_clause
```

```
;
```

```
subshell      : '(' compound_list ')'
```

```
;
```

```

compound_list  : linebreak term
                | linebreak term separator
                ;

term           : term separator and_or
                |
                and_or
                ;

for_clause     : For name                                do_group
                | For name                                sequential_sep do_group
                | For name linebreak in                    sequential_sep do_group
                | For name linebreak in wordlist sequential_sep do_group
                ;

name           : NAME                                    /* Apply rule 5 */
                ;

in            : In                                       /* Apply rule 6 */
                ;

wordlist       : wordlist WORD
                |
                WORD
                ;

case_clause    : Case WORD linebreak in linebreak case_list      Esac
                | Case WORD linebreak in linebreak case_list_ns Esac
                | Case WORD linebreak in linebreak                Esac
                ;

case_list_ns   : case_list case_item_ns
                |
                case_item_ns
                ;

case_list      : case_list case_item
                |
                case_item
                ;

case_item_ns   :      pattern ')' linebreak
                |      pattern ')' compound_list
                | '(' pattern ')' linebreak
                | '(' pattern ')' compound_list
                ;

case_item      :      pattern ')' linebreak      DSEMI linebreak
                |      pattern ')' compound_list DSEMI linebreak
                | '(' pattern ')' linebreak      DSEMI linebreak
                | '(' pattern ')' compound_list DSEMI linebreak
                ;

pattern        :      WORD                                /* Apply rule 4 */
                | pattern '|' WORD                        /* Do not apply rule 4 */
                ;

if_clause      : If compound_list Then compound_list else_part Fi
                | If compound_list Then compound_list          Fi
                ;

else_part      : Elif compound_list Then compound_list
                | Elif compound_list Then compound_list else_part
                | Else compound_list
                ;

while_clause   : While compound_list do_group

```

```

;
until_clause      : Until compound_list do_group
;
function_definition : fname '(' ')' linebreak function_body
;
function_body      : compound_command          /* Apply rule 9 */
                    | compound_command redirect_list /* Apply rule 9 */
;
fname              : NAME                      /* Apply rule 8 */
;
brace_group        : Lbrace compound_list Rbrace
;
do_group           : Do compound_list Done      /* Apply rule 6 */
;
simple_command      : cmd_prefix cmd_word cmd_suffix
                    | cmd_prefix cmd_word
                    | cmd_prefix
                    | cmd_name cmd_suffix
                    | cmd_name
;
cmd_name           : WORD                      /* Apply rule 7a */
;
cmd_word           : WORD                      /* Apply rule 7b */
;
cmd_prefix         :          io_redirect
                    | cmd_prefix io_redirect
                    |          ASSIGNMENT_WORD
                    | cmd_prefix ASSIGNMENT_WORD
;
cmd_suffix         :          io_redirect
                    | cmd_suffix io_redirect
                    |          WORD
                    | cmd_suffix WORD
;
redirect_list      :          io_redirect
                    | redirect_list io_redirect
;
io_redirect        :          io_file
                    | IO_NUMBER io_file
                    |          io_here
                    | IO_NUMBER io_here
;
io_file            : '<'      filename
                    | LESSAND filename
                    | '>'      filename
                    | GREATAND filename
                    | DGREAT  filename
                    | LESSGREAT filename
                    | CLOBBER  filename

```



```

;
filename      : WORD                      /* Apply rule 2 */
;
io_here       : DLESS      here_end
               | DLESSDASH here_end
;
here_end      : WORD                      /* Apply rule 3 */
;
newline_list  : NEWLINE
               | newline_list NEWLINE
;
linebreak     : newline_list
               | /* empty */
;
separator_op  : '&'
               | ';'
;
separator     : separator_op linebreak
               | newline_list
;
sequential_sep : ';' linebreak
               | newline_list
;

```

## 2.11. Signals and Error Handling



If job control is disabled (see the description of [set -m](#)) when the shell executes an asynchronous list, the commands in the list shall inherit from the shell a signal action of ignored (SIG\_IGN) for the SIGINT and SIGQUIT signals. In all other cases, commands executed by the shell shall inherit the same signal actions as those inherited by the shell from its parent unless a signal action is modified by the [trap](#) special built-in (see [trap](#))

When a signal for which a trap has been set is received while the shell is waiting for the completion of a utility executing a foreground command, the trap associated with that signal shall not be executed until after the foreground command has completed. When the shell is waiting, by means of the [wait](#) utility, for asynchronous commands to complete, the reception of a signal for which a trap has been set shall cause the [wait](#) utility to return immediately with an exit status >128, immediately after which the trap associated with that signal shall be taken.

If multiple signals are pending for the shell for which there are associated trap actions, the order of execution of trap actions is unspecified.

## 2.12. Shell Execution Environment

A shell execution environment consists of the following:

- Open files inherited upon invocation of the shell, plus open files controlled by [exec](#)
- Working directory as set by [cd](#)
- File creation mask set by [umask](#)
- [\[XSI\]](#)  File size limit as set by [ulimit](#) 
- Current traps set by [trap](#)
- Shell parameters that are set by variable assignment (see the [set](#) special built-in) or from the System Interfaces volume of POSIX.1-2017 environment inherited by the shell when it begins (see the [export](#) special

built-in)

- Shell functions; see [Function Definition Command](#)
- Options turned on at invocation or by [set](#)
- Process IDs of the last commands in asynchronous lists known to this shell environment; see [Asynchronous Lists](#)
- Shell aliases; see [Alias Substitution](#)

Utilities other than the special built-ins (see [Special Built-In Utilities](#)) 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 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 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 that are not being ignored shall be set to the default action. 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 multi-command 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.

## 2.13. Pattern Matching Notation

The pattern matching notation described in this section 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 XBD [Regular Expressions](#). For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation, modified to account for the differences.

### 2.13.1 Patterns Matching a Single Character

The following patterns matching a single character shall match a single character: ordinary characters, special pattern characters, and pattern bracket expressions. The pattern bracket expression also shall match a single collating element. A `<backslash>` character shall escape the following character. The escaping `<backslash>` shall be discarded. If a pattern ends with an unescaped `<backslash>`, it is unspecified whether the pattern does not match anything or the pattern is treated as invalid.

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 [Quoting](#) that require quoting, and the following three special pattern characters. Matching shall be based on the bit pattern used for encoding the character, not on the graphic representation of the character. If any character (ordinary, shell special, or pattern special) is quoted, that pattern shall match the character itself. The shell special characters always require quoting.

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

?

A `<question-mark>` is a pattern that shall match any character.

\*

An <asterisk> is a pattern that shall match multiple characters, as described in [Patterns Matching Multiple Characters](#).

[

If an open bracket introduces a bracket expression as in XBD [RE Bracket Expression](#), except that the <exclamation-mark> character ( ' ! ' ) shall replace the <circumflex> character ( ' ^ ' ) in its role in a non-matching list in the regular expression notation, it shall introduce a pattern bracket expression. A bracket expression starting with an unquoted <circumflex> character produces unspecified results. Otherwise, ' [ ' shall match the character itself.

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 one of the `exec` functions as defined in the System Interfaces volume of POSIX.1-2017, 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> character. This escaping <backslash> is discarded. The sequence "\\\" represents 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.

### 2.13.2 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 <asterisk> characters 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.

### 2.13.3 Patterns Used for Filename Expansion

The rules described so far in [Patterns Matching a Single Character](#) and [Patterns Matching Multiple Characters](#) 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 <slash> characters in the pattern; it shall neither be matched by the <asterisk> or <question-mark> special characters nor by a bracket expression. <slash> characters in the pattern shall be identified before bracket expressions; thus, a <slash> cannot be included in a pattern bracket expression used for filename expansion. If a <slash> character is found following an unescaped <left-square-bracket> character before a corresponding <right-square-bracket> is found, the open bracket shall be treated as an ordinary character. For example, the pattern "a[b/c]d" does not match such pathnames as **abd** or **a/d**. It only matches a pathname of literally **a[b/c]d**.
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
  - A bracket expression containing a non-matching 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 shall be matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character shall require read permission in the directory containing that component. Any component, except the last, that does not contain a pattern character shall require search permission. For example, given the pattern:

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

search permission is needed for directories **/** and **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 this collating sequence does not have a total ordering of all characters (see XBD [LC\\_COLLATE](#)), any filenames or pathnames that collate equally should be further compared byte-by-byte using the collating sequence for the POSIX locale.

**Note:**

A future version of this standard may require the byte-by-byte further comparison described above.

If the pattern contains an open bracket ( `' [ '` ) that does not introduce a bracket expression as in XBD [RE Bracket Expression](#), it is unspecified whether other unquoted pattern matching characters within the same slash-delimited component of the pattern retain their special meanings or are treated as ordinary characters. For example, the pattern `"a* [ /b*"` may match all filenames beginning with `' b '` in the directory `"a* [ "` or it may match all filenames beginning with `' b '` in all directories with names beginning with `' a '` and ending with `' [ '`.

If the pattern does not match any existing filenames or pathnames, the pattern string shall be left unchanged.

## 2.14. 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 may 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. An error in a special built-in utility may cause a shell executing that utility to abort, while an error in a regular built-in utility shall not cause a shell executing that utility to abort. (See [Consequences of Shell Errors](#) for the consequences of errors on interactive and non-interactive shells.) If a special built-in utility encountering an error does not abort the shell, its exit value shall be non-zero.
2. As described in [Simple Commands](#), variable assignments preceding the invocation of a special built-in utility remain in effect after the built-in completes; this shall not be the case with a regular built-in or other utility.

The special built-in utilities in this section need not be provided in a manner accessible via the `exec` family of functions defined in the System Interfaces volume of POSIX.1-2017.

Some of the special built-ins are described as conforming to XBD [Utility Syntax Guidelines](#). For those that are not, the requirement in [Utility Description Defaults](#) that `" - - "` be recognized as a first argument to be discarded does not apply and a conforming application shall not use that argument.

[<<< Previous](#)

[Home](#)

[Next >>>](#)

[return to top of page](#)

---

UNIX ® is a registered Trademark of The Open Group.  
 POSIX ™ is a Trademark of The IEEE.  
 Copyright © 2001-2018 IEEE and The Open Group, All Rights Reserved  
 [ [Main Index](#) | [XBD](#) | [XSH](#) | [XCU](#) | [XRAT](#) ]

---

### NAME

`break` - exit from `for`, `while`, or `until` loop

## SYNOPSIS

`break` [*n*]

## DESCRIPTION

If *n* is specified, the [break](#) utility shall exit from the *n*th enclosing **for**, **while**, or **until** loop. If *n* is not specified, [break](#) shall behave as if *n* was specified as 1. Execution shall continue with the command immediately following the exited loop. The value of *n* is a positive decimal integer. If *n* is greater than the number of enclosing loops, the outermost enclosing loop shall be exited. If there is no enclosing loop, the behavior is unspecified.

A loop shall enclose a *break* or *continue* command if the loop lexically encloses the command. A loop lexically encloses a *break* or *continue* command if the command is:

- Executing in the same execution environment (see [Shell Execution Environment](#)) as the compound-list of the loop's do-group (see [Shell Grammar Rules](#)), and
- Contained in a compound-list associated with the loop (either in the compound-list of the loop's do-group or, if the loop is a **while** or **until** loop, in the compound-list following the **while** or **until** reserved word), and
- Not in the body of a function whose function definition command (see [Function Definition Command](#)) is contained in a compound-list associated with the loop.

If *n* is greater than the number of lexically enclosing loops and there is a non-lexically enclosing loop in progress in the same execution environment as the *break* or *continue* command, it is unspecified whether that loop encloses the command.

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

Not used.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

- 0 Successful completion.
- >0 The *n* value was not an unsigned decimal integer greater than or equal to 1.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

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

The results of running the following example are unspecified: there are two loops in progress when the [break](#) command is executed, and they are in the same execution environment, but neither loop is lexically enclosing the [break](#) command. (There are no loops lexically enclosing the [continue](#) commands, either.)

```
foo() {
    for j in 1 2; do
        echo 'break 2' >/tmp/do_break
        echo " sourcing /tmp/do_break ($j)..."
        # the behavior of the break from running the following command
        # results in unspecified behavior:
        . /tmp/do_break
    done
}
```

```

do_continue() { continue 2; }
echo "  running do_continue ($j)..."
# the behavior of the continue in the following function call
# results in unspecified behavior (if execution reaches this
# point):
do_continue

trap 'continue 2' USR1
echo "  sending SIGUSR1 to self ($j)..."
# the behavior of the continue in the trap invoked from the
# following signal results in unspecified behavior (if
# execution reaches this point):
kill -s USR1 $$
sleep 1
done
}
for i in 1 2; do
    echo "running foo ($i)..."
    foo
done

```

## RATIONALE

In early proposals, consideration was given to expanding the syntax of [break](#) and [continue](#) to refer to a label associated with the appropriate loop as a preferable alternative to the *n* method. However, this volume of POSIX.1-2017 does reserve the name space 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.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY



## Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

## Issue 7

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0046 [842] is applied.

*End of informative text.*

---

[<<< Previous](#)[Home](#)[Next >>>](#)

---

## NAME

colon - null utility

## SYNOPSIS

: [*argument...*]

## DESCRIPTION

This utility shall only expand command *arguments*. It 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.

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

Not used.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

Zero.

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## **APPLICATION USAGE**

None.

## **EXAMPLES**

```
: ${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*.

## **RATIONALE**

None.

## **FUTURE DIRECTIONS**

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

continue - continue for, while, or until loop

## SYNOPSIS

continue [*n*]

## DESCRIPTION

If *n* is specified, the [continue](#) utility shall return to the top of the *n*th enclosing **for**, **while**, or **until** loop. If *n* is not specified, [continue](#) shall behave as if *n* was specified as 1. Returning to the top of the loop involves repeating the condition list of a **while** or **until** loop or performing the next assignment of a **for** loop, and re-executing the loop if appropriate.

The value of *n* is a positive decimal integer. If *n* is greater than the number of enclosing loops, the outermost enclosing loop shall be used. If there is no enclosing loop, the behavior is unspecified.

The meaning of "enclosing" shall be as specified in the description of the [break](#) utility.

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

Not used.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

- 0 Successful completion.
- >0 The  $n$  value was not an unsigned decimal integer greater than or equal to 1.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

```
for i in *
do
    if test -d "$i"
    then continue
    fi
    printf '"%s" is not a directory.\n' "$i"
done
```

## RATIONALE

None.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

The example is changed to use the [printf](#) utility rather than [echo](#).

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0046 [842] is applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

dot - execute commands in the current environment

## SYNOPSIS

`. file`

## DESCRIPTION

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 non-interactive shell shall abort; an interactive shell shall write a diagnostic message to standard error, but this condition shall not be considered a syntax error.

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## **STDIN**

Not used.

## **INPUT FILES**

See the DESCRIPTION.

## **ENVIRONMENT VARIABLES**

See the DESCRIPTION.

## **ASYNCHRONOUS EVENTS**

Default.

## **STDOUT**

Not used.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

If no readable file was found or if the commands in the file could not be parsed, and the shell is interactive (and therefore does not abort; see [Consequences of Shell Errors](#)), the exit status shall be non-zero. Otherwise, return the value of the last command executed, or a zero exit status if no command is executed.

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## **APPLICATION USAGE**

None.

## **EXAMPLES**

```
cat foobar
foo=hello bar=world
```

```
. ./foobar
echo $foo $bar
hello world
```

## RATIONALE

Some older implementations searched the current directory for the *file*, even if the value of *PATH* disallowed it. This behavior was omitted from this volume of POSIX.1-2008 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 positional parameters. This is a valid extension that allows a [dot](#) script to behave identically to a function.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#), [return](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-164 is applied.

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0038 [114] and XCU/TC1-2008/0039 [214] are applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

eval - construct command by concatenating arguments

## SYNOPSIS

```
eval [argument...]
```

## DESCRIPTION

The [eval](#) utility shall construct a command by concatenating *arguments* together, separating each with a <space> character. The constructed command shall be read and executed by the shell.



## **OPTIONS**

None.

## **OPERANDS**

See the DESCRIPTION.

## **STDIN**

Not used.

## **INPUT FILES**

None.

## **ENVIRONMENT VARIABLES**

None.

## **ASYNCHRONOUS EVENTS**

Default.

## **STDOUT**

Not used.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

If there are no *arguments*, 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 *arguments* separated by <space> characters, or a non-zero exit status if the concatenation could not be parsed as a command and the shell is interactive (and therefore did not abort).

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

Since [eval](#) is not required to recognize the " - " end of options delimiter, in cases where the argument(s) to [eval](#) might begin with ' - ' it is recommended that the first argument is prefixed by a string that will not alter the commands to be executed, such as a <space> character:

```
eval " $commands"
```

or:

```
eval " $(some_command) "
```

## EXAMPLES

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

## RATIONALE

This standard allows, but does not require, [eval](#) to recognize " - ". Although this means applications cannot use " - " to protect against options supported as an extension (or errors reported for unsupported options), the nature of the [eval](#) utility is such that other means can be used to provide this protection (see APPLICATION USAGE above).

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0040 [114], XCU/TC1-2008/0041 [163], and XCU/TC1-2008/0042 [163] are applied.

End of informative text.

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

exec - execute commands and open, close, or copy file descriptors

## SYNOPSIS

exec [*command* [*argument...*]]

## DESCRIPTION

The [exec](#) utility shall open, close, and/or copy 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 greater than 2 are opened with associated redirection statements, it is unspecified whether those file descriptors remain open when the shell invokes another utility. Scripts concerned that child shells could misuse open file descriptors can always close them explicitly, as shown in one of the following examples.

If [exec](#) is specified with *command*, it shall replace the shell with *command* without creating a new process. If *arguments* are specified, they shall be arguments to *command*. Redirection affects the current shell execution environment.

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

Not used.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

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

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

Open *readfile* as file descriptor 3 for reading:

```
exec 3< readfile
```

Open *writefile* as file descriptor 4 for writing:

```
exec 4> writefile
```

Make file descriptor 5 a copy of file descriptor 0:

```
exec 5<&0
```

Close file descriptor 3:

```
exec 3<&-
```

Cat the file **maggie** by replacing the current shell with the [cat](#) utility:

```
exec cat maggie
```

## RATIONALE

Most historical implementations were not conformant in that:

```
foo=bar exec cmd
```

did not pass **foo** to **cmd**.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

exit - cause the shell to exit

## SYNOPSIS

```
exit [n]
```

## DESCRIPTION

The [exit](#) utility shall cause the shell to exit from its current execution environment with the exit status specified by the unsigned decimal integer *n*. If the current execution environment is a subshell environment, the shell shall exit from the subshell environment with the specified exit status and continue in the environment from which that subshell environment was invoked; otherwise, the shell utility shall terminate with the specified exit status. 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.

## OPTIONS

None.

## **OPERANDS**

See the DESCRIPTION.

## **STDIN**

Not used.

## **INPUT FILES**

None.

## **ENVIRONMENT VARIABLES**

None.

## **ASYNCHRONOUS EVENTS**

Default.

## **STDOUT**

Not used.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

The exit status shall be  $n$ , if specified, except that the behavior is unspecified if  $n$  is not an unsigned decimal integer or is greater than 255. 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, the last command is considered to be the command that executed immediately preceding the [trap](#) action.

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## **APPLICATION USAGE**

None.

## EXAMPLES

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```

Propagate error handling from within a subshell:

```
(  
    command1 || exit 1  
    command2 || exit 1  
    exec command3  
) > outputfile || exit 1  
echo "outputfile created successfully"
```

## RATIONALE

As explained in other sections, 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.

The behavior of [exit](#) when given an invalid argument or unknown option is unspecified, because of differing practices in the various historical implementations. A value larger than 255 might be truncated by the shell, and be unavailable even to a parent process that uses [waitid\(\)](#) to get the full exit value. It is recommended that implementations that detect any usage error should cause a non-zero exit status (or, if the shell is interactive and the error does not cause the shell to abort, store a non-zero value in "\$?" ), but even this was not done historically in all shells.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections



use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

## Issue 7

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0047 [717], XCU/TC2-2008/0048 [960], XCU/TC2-2008/0049 [717], and XCU/TC2-2008/0050 [960] are applied.

*End of informative text.*

---

[<<< Previous](#)[Home](#)[Next >>>](#)

---

## NAME

`export` - set the export attribute for variables

## SYNOPSIS

```
export name[=word] ...
```

```
export -p
```

## DESCRIPTION

The shell shall give the [export](#) attribute to the variables corresponding to the specified *names*, which shall cause them to be in the environment of subsequently executed commands. If the name of a variable is followed by = *word*, then the value of that variable shall be set to *word*.

The [export](#) special built-in shall support XBD [Utility Syntax Guidelines](#).

When **-p** is specified, [export](#) shall write to the standard output the names and values of all exported variables, in the following format:

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

if *name* is set, and:

```
"export %s\n", <name>
```

if *name* is unset.

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same exporting results, except:

1. Read-only variables with values cannot be reset.
2. Variables that were unset at the time they were output need not be reset to the unset state if a value is assigned to the variable between the time the state was saved and the time at which the saved output is reinput to the shell.

When no arguments are given, the results are unspecified.

## OPTIONS

See the DESCRIPTION.

## **OPERANDS**

See the DESCRIPTION.

## **STDIN**

Not used.

## **INPUT FILES**

None.

## **ENVIRONMENT VARIABLES**

None.

## **ASYNCHRONOUS EVENTS**

Default.

## **STDOUT**

See the DESCRIPTION.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

- 0 All *name* operands were successfully exported.
- >0 At least one *name* could not be exported, or the **-p** option was specified and an error occurred.

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## **APPLICATION USAGE**

Note that, unless *X* was previously marked readonly, the value of "\$?" after:

```
export X=$(false)
```

will be 0 (because [export](#) successfully set *X* to the empty string) and that execution continues, even if [set -e](#) is in effect. In order to detect command substitution failures, a user must separate the assignment from the export, as in:

```
X=$(false)
export X
```

## 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
```

## RATIONALE

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 historical 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 example, a [dot](#) script.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

XBD [Utility Syntax Guidelines](#)

## CHANGE HISTORY

### Issue 6

IEEE PASC Interpretation 1003.2 #203 is applied, clarifying the format when a variable is unset.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/6 is applied, adding the following text to the end of the first paragraph of the DESCRIPTION: "If the name of a variable is followed by = *word*, then the value of that variable shall be set to *word*.". The reason for this change is that the SYNOPSIS for [export](#) includes:

```
export name[=word]...
```

but the meaning of the optional "= *word*" is never explained in the text.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0043 [352] is applied.

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0051 [654] and XCU/TC2-2008/0052 [960] are applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

readonly - set the readonly attribute for variables

## SYNOPSIS

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

```
readonly -p
```

## DESCRIPTION

The variables whose *names* are specified shall be given the [readonly](#) attribute. The values of variables with the [readonly](#) attribute cannot be changed by subsequent assignment, nor can those variables be unset by the [unset](#) utility. If the name of a variable is followed by = *word*, then the value of that variable shall be set to *word*.

The [readonly](#) special built-in shall support XBD [Utility Syntax Guidelines](#).

When **-p** is specified, [readonly](#) writes to the standard output the names and values of all read-only variables, in the following format:

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

if *name* is set, and

```
"readonly %s\n", <name>
```

if *name* is unset.

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same value and *readonly* attribute-setting results in a shell

execution environment in which:

1. Variables with values at the time they were output do not have the *readonly* attribute set.
2. Variables that were unset at the time they were output do not have a value at the time at which the saved output is reinput to the shell.

When no arguments are given, the results are unspecified.

## OPTIONS

See the DESCRIPTION.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

See the DESCRIPTION.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

- 0  
All *name* operands were successfully marked readonly.
- >0  
At least one *name* could not be marked readonly, or the **-p** option was specified and an error

occurred.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

```
readonly HOME PWD
```

## RATIONALE

Some historical shells preserve the *readonly* attribute across separate invocations. This volume of POSIX.1-2008 allows this behavior, but does not require it.

The **-p** option allows portable access to the values that can be saved and then later restored using, for example, a [dot](#) script. Also see the RATIONALE for [export](#) for a description of the no-argument and **-p** output cases and a related example.

Read-only functions were considered, but they were omitted as not being historical practice or particularly useful. Furthermore, functions must not be read-only 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 real intent of the user) of administrative or security-relevant (or security-conscious) shell scripts.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

XBD [Utility Syntax Guidelines](#)

## CHANGE HISTORY

### Issue 6

IEEE PASC Interpretation 1003.2 #203 is applied, clarifying the format when a variable is unset.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/7 is applied, adding the following text to the end of the first paragraph of the DESCRIPTION: "If the name of a variable is followed by = *word*, then the value of that variable shall be set to *word*". The reason for this change is that the SYNOPSIS for [readonly](#) includes:

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

but the meaning of the optional "*= word*" is never explained in the text.

## Issue 7

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0052 [960] is applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

return - return from a function or dot script

## SYNOPSIS

```
return [n]
```

## DESCRIPTION

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

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

Not used.



## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## 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 *n* is not an unsigned decimal integer, or is greater than 255, the results are unspecified. When [return](#) is executed in a [trap](#) action, the last command is considered to be the command that executed immediately preceding the [trap](#) action.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

None.

## RATIONALE

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 under [exit](#).

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Function Definition Command](#), [Special Built-In Utilities](#), [dot](#)

## CHANGE HISTORY

## Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

## Issue 7

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0044 [214] and XCU/TC1-2008/0045 [214] are applied.

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0052 [960] is applied.

*End of informative text.*

---

[<<< Previous](#)[Home](#)[Next >>>](#)

---

## NAME

set - set or unset options and positional parameters

## SYNOPSIS

```
set [-abCefhmnux] [-o option] [argument...]
```

```
set [+abCefhmnux] [+o option] [argument...]
```

```
set -- [argument...]
```

```
set -o
```

```
set +o
```

## DESCRIPTION

If no *options* or *arguments* 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; see the description of shell quoting in [Quoting](#). The output shall be suitable for reinput to the shell, setting or resetting, as far as possible, the variables that are currently set; read-only variables cannot be reset.

When options are specified, they shall set or unset attributes of the shell, as described below. When *arguments* are specified, they cause positional parameters to be set or unset, as described below. Setting or unsetting attributes and positional parameters are not necessarily related actions, but they can be combined in a single invocation of [set](#).

The [set](#) special built-in shall support XBD [Utility Syntax Guidelines](#) except that options can be specified with either a leading <hyphen-minus> (meaning enable the option) or <plus-sign> (meaning disable it) unless otherwise specified.

Implementations shall support the options in the following list in both their <hyphen-minus> and <plus-sign> forms. These options can also be specified as options to [sh](#).

**-a**

When this option is on, the *export* attribute shall be set for each variable to which an assignment is performed; see XBD [Variable Assignment](#). If the assignment precedes a utility name in a command, the *export* attribute shall not persist in the current execution environment after the utility completes, with the exception that preceding one of the special built-in utilities causes 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, the *export* attribute shall persist until the variable is unset.

**-b**

This option shall be supported if the implementation supports the User Portability Utilities option. It shall cause the shell to notify the user asynchronously of background job completions. The following message is written to standard error:

```
"[%d]%c %s%s\n", <job-number>, <current>, <status>, <job-name>
```

where the fields shall be as follows:

<current>

The character '+' identifies the job that would be used as a default for the [fg](#) or [bg](#) utilities; this job can also be specified using the *job\_id* "%+" or "%%". The character '-' identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job\_id* "%-". For other jobs, this field is a <space>. At most one job can be identified with '+' and at most one job can be identified with '-'. If there is any suspended job, then the current job shall be a suspended job. If there are at least two suspended jobs, then the previous job also shall be a suspended job.

<job-number>

A number that can be used to identify the process group to the [wait](#), [fg](#), [bg](#), and [kill](#) utilities. Using these utilities, the job can be identified by prefixing the job number with '% '.

<status>

Unspecified.

<job-name>

Unspecified.

When the shell notifies the user a job has been completed, it may remove the job's process ID from the list of those known in the current shell execution environment; see [Asynchronous Lists](#). Asynchronous notification shall not be enabled by default.

**-C**

(Uppercase C.) Prevent existing files from being overwritten by the shell's '>' redirection operator (see [Redirecting Output](#)); the ">|" redirection operator shall override this *noclobber* option for an individual file.

**-e**

When this option is on, when any command fails (for any of the reasons listed in [Consequences of Shell Errors](#) or by returning an exit status greater than zero), the shell immediately shall exit, as if by executing the [exit](#) special built-in utility with no arguments, with the following exceptions:

1. The failure of any individual command in a multi-command pipeline shall not cause the shell to exit. Only the failure of the pipeline itself shall be considered.
2. The **-e** setting shall be ignored when executing the compound list following the **while**, **until**, **if**, or **elif** reserved word, a pipeline beginning with the **!** reserved word, or any command of an AND-OR list other than the last.
3. If the exit status of a compound command other than a subshell command was the result of a failure while **-e** was being ignored, then **-e** shall not apply to this command.

This requirement applies to the shell environment and each subshell environment separately. For example, in:

```
set -e; (false; echo one) | cat; echo two
```

the [false](#) command causes the subshell to exit without executing `echo one`; however, `echo two` is executed because the exit status of the pipeline `(false; echo one) | cat` is zero.

- f**  
The shell shall disable pathname expansion.
- h**  
Locate and remember utilities invoked by functions as those functions are defined (the utilities are normally located when the function is executed).
- m**  
This option shall be supported if the implementation supports the User Portability Utilities option. All jobs shall be run in their own process groups. Immediately before the shell issues a prompt after completion of the background job, a message reporting the exit status of the background job shall be written to standard error. If a foreground job stops, the shell shall write a message to standard error to that effect, formatted as described by the [jobs](#) utility. In addition, if a job changes status other than exiting (for example, if it stops for input or output or is stopped by a SIGSTOP signal), the shell shall write a similar message immediately prior to writing the next prompt. This option is enabled by default for interactive shells.
- n**  
The shell shall read commands but does not execute them; this can be used to check for shell script syntax errors. An interactive shell may ignore this option.
- o**  
Write the current settings of the options to standard output in an unspecified format.
- +o**  
Write the current option settings to standard output in a format that is suitable for reinput to the shell as commands that achieve the same options settings.
- o option**  
This option is supported if the system supports the User Portability Utilities option. It shall set various options, many of which shall be equivalent to the single option letters. The following values of *option* shall be supported:
  - allexport*  
Equivalent to **-a**.
  - errexit*  
Equivalent to **-e**.
  - ignoreeof*  
Prevent an interactive shell from exiting on end-of-file. This setting prevents accidental logouts when `<control>-D` is entered. A user shall explicitly [exit](#) to leave the interactive shell.
  - monitor*  
Equivalent to **-m**. This option is supported if the system supports the User Portability Utilities option.
  - noclobber*  
Equivalent to **-C** (uppercase C).
  - noglob*  
Equivalent to **-f**.
  - noexec*  
Equivalent to **-n**.
  - nolog*  
Prevent the entry of function definitions into the command history; see [Command History List](#).
  - notify*  
Equivalent to **-b**.
  - nounset*  
Equivalent to **-u**.
  - verbose*  
Equivalent to **-v**.
  - vi*  
Allow shell command line editing using the built-in [vi](#) editor. Enabling [vi](#) mode shall disable any other command line editing mode provided as an implementation extension.  
  
It need not be possible to set [vi](#) mode on for certain block-mode terminals.
  - xtrace*  
Equivalent to **-x**.
- u**

When the shell tries to expand an unset parameter other than the '@' and '\*' special parameters, it shall write a message to standard error and the expansion shall fail with the consequences specified in [Consequences of Shell Errors](#).

**-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. It is unspecified whether the command that turns tracing off is traced.

The default for all these options shall be off (unset) unless stated otherwise in the description of the option or unless the shell was invoked with them on; see [sh](#).

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. All positional parameters shall be unset before any new values are assigned.

If the first argument is '-', the results are unspecified.

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* shall unset all positional parameters and set the special parameter '#' to zero.

## OPTIONS

See the DESCRIPTION.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

See the DESCRIPTION.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

- 0 Successful completion.
- >0 An invalid option was specified, or an error occurred.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

Application writers should avoid relying on [set -e](#) within functions. For example, in the following script:

```
set -e
start() {
    some_server
    echo some_server started successfully
}
start || echo ">&2 some_server failed"
```

the **-e** setting is ignored within the function body (because the function is a command in an AND-OR list other than the last). Therefore, if `some_server` fails, the function carries on to echo "some\_server started successfully", and the exit status of the function is zero (which means "some\_server failed" is not output).

## 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 it begins with ' - ' or ' + ' :

```
set -- "$x"
```

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

```
set -- $x
```

## RATIONALE

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 issue it without arguments deliberately; if it were issued as, for example:

```
set -- "$@"
```

and there were in fact no arguments resulting from "\$@", unsetting the parameters would have no result.

The [set](#) + form in early proposals was omitted as being an unnecessary duplication of [set](#) alone and not widespread historical practice.

The *noclobber* option was changed to allow [set](#) -C as well as the [set](#) -o *noclobber* option. The single-letter version was added so that the historical "\$ - " paradigm would not be broken; see [Special Parameters](#).

The description of the -e option is intended to match the behavior of the 1988 version of the KornShell.

The -h flag is related to command name hashing. See [hash](#).

The following [set](#) flags were omitted intentionally with the following rationale:

### -k

The -k flag was originally added by the author of the Bourne shell to make it easier for users of pre-release 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* differs. The reference 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 while *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 the historical practice of using *+option* was satisfactory and that there was no compelling reason to modify such widespread historical practice.

The *-o* option was adopted from the KornShell to address user needs. In addition to its generally friendly interface, *-o* is needed to provide the *vi* command line editing mode, for which historical practice yields no single-letter option name. (Although it might have been possible to invent such a letter, it was recognized that other editing modes would be developed and *-o* provides ample name space for describing such extensions.)

Historical implementations are inconsistent in the format used for *-o* option status reporting. The *+o* format without an option-argument was added to allow portable access to the options that can be saved and then later restored using, for instance, a dot script.

Historically, *sh* did trace the command *set +x*, but *ksh* did not.

The *ignoreeof* setting prevents accidental logouts when the end-of-file character (typically <control>-D) is entered. A user shall explicitly *exit* to leave the interactive shell.

The *set -m* option was added to apply only to the UPE because it applies primarily to interactive use, not shell script applications.

The ability to do asynchronous notification became available in the 1988 version of the KornShell. To have it occur, the user had to issue the command:

```
trap "jobs -n" CLD
```

The C shell provides two different levels of an asynchronous notification capability. The environment variable *notify* is analogous to what is done in *set -b* or *set -o notify*. When set, it notifies the user immediately of background job completions. When unset, this capability is turned off.

The other notification ability comes through the built-in utility *notify*. The syntax is:

```
notify [%job ... ]
```

By issuing *notify* with no operands, it causes the C shell to notify the user asynchronously when the state of the current job changes. If given operands, *notify* asynchronously informs the user of changes in the states of the specified jobs.

To add asynchronous notification to the POSIX shell, neither the KornShell extensions to *trap*, nor the C shell *notify* environment variable seemed appropriate (*notify* is not a proper POSIX environment variable name).

The *set -b* option was selected as a compromise.

The *notify* built-in was considered to have more functionality than was required for simple asynchronous notification.

Historically, some shells applied the *-u* option to all parameters including *\$@* and *\$\**. The standard developers felt that this was a misfeature since it is normal and common for *\$@* and *\$\** to be used in shell scripts regardless of whether they were passed any arguments. Treating these uses as an error when no arguments are passed reduces the value of *-u* for its intended purpose of finding spelling mistakes in variable names and uses of unset positional parameters.

## FUTURE DIRECTIONS

None.



## SEE ALSO

[Special Built-In Utilities](#), [hash](#)

XBD [Variable Assignment](#), [Utility Syntax Guidelines](#)

## CHANGE HISTORY

### Issue 6

The obsolescent [set](#) command name followed by ' - ' has been removed.

The following new requirements on POSIX implementations derive from alignment with the Single UNIX Specification:

- The *nolog* option is added to [set](#) -o.

IEEE PASC Interpretation 1003.2 #167 is applied, clarifying that the options default also takes into account the description of the option.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/8 is applied, changing the square brackets in the example in RATIONALE to be in bold, which is the typeface used for optional items.

### Issue 7

Austin Group Interpretation 1003.1-2001 #027 is applied, clarifying the behavior if the first argument is ' - '.

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

XSI shading is removed from the -h functionality.

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0046 [52], XCU/TC1-2008/0047 [155,280], XCU/TC1-2008/0048 [52], XCU/TC1-2008/0049 [52], and XCU/TC1-2008/0050 [155,430] are applied.

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0053 [584], XCU/TC2-2008/0054 [717], XCU/TC2-2008/0055 [717], and XCU/TC2-2008/0056 [960] are applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

shift - shift positional parameters

## SYNOPSIS

shift [*n*]

## DESCRIPTION

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 on. The parameters represented by the numbers "\$#" down to "\$#-*n*+1" shall be unset, and the parameter

'#' is 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 are not changed.

## **OPTIONS**

None.

## **OPERANDS**

See the DESCRIPTION.

## **STDIN**

Not used.

## **INPUT FILES**

None.

## **ENVIRONMENT VARIABLES**

None.

## **ASYNCHRONOUS EVENTS**

Default.

## **STDOUT**

Not used.

## **STDERR**

The standard error shall be used only for diagnostic messages.

## **OUTPUT FILES**

None.

## **EXTENDED DESCRIPTION**

None.

## **EXIT STATUS**

If the  $n$  operand is invalid or is greater than "\$#", this may be considered a syntax error and a non-interactive shell may exit; if the shell does not exit in this case, a non-zero exit status shall be returned. Otherwise, zero shall be returned.

## **CONSEQUENCES OF ERRORS**

Default.

---

*The following sections are informative.*

## **APPLICATION USAGE**

None.

## **EXAMPLES**

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

## **RATIONALE**

None.

## **FUTURE DIRECTIONS**

None.

## **SEE ALSO**

[Special Built-In Utilities](#)

## **CHANGE HISTORY**

### **Issue 6**

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### **Issue 7**

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0051 [459] is applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## **NAME**

times - write process times

## **SYNOPSIS**

```
times
```

## **DESCRIPTION**

The [times](#) utility shall write the accumulated user and system times for the shell and for all of its child processes, in the following POSIX locale format:

```
"%dm%fs %dm%fs\n%dm%fs %dm%fs\n", <shell user minutes>,  
    <shell user seconds>, <shell system minutes>,  
    <shell system seconds>, <children user minutes>,  
    <children user seconds>, <children system minutes>,  
    <children system seconds>
```

The four pairs of times shall correspond to the members of the  [<sys/times.h> tms](#<sys/times.h> tms) structure (defined in XBD [Headers](#)) as returned by [times\(\)](#): *tms\_utime*, *tms\_stime*, *tms\_cutime*, and *tms\_cstime*, respectively.

## OPTIONS

None.

## OPERANDS

None.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

See the DESCRIPTION.

## STDERR

The standard error shall be used only for diagnostic messages.

## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

0	Successful completion.
>0	An error occurred.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

```
$ times
0m0.43s 0m1.11s
8m44.18s 1m43.23s
```

## RATIONALE

The [times](#) special built-in from the Single UNIX Specification is now required for all conforming shells.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

XBD [<sys/times.h>](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/9 is applied, changing text in the DESCRIPTION from: "Write the accumulated user and system times for the shell and for all of its child processes ..." to: "The [times](#) utility shall write the accumulated user and system times for the shell and for all of its child processes ...".

### Issue 7

POSIX.1-2008, Technical Corrigendum 2, XCU/TC2-2008/0056 [960] is applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

trap - trap signals

## SYNOPSIS

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

## DESCRIPTION

If the first operand is an unsigned decimal integer, the shell shall treat all operands as conditions, and shall reset each condition to the default value. Otherwise, if there are operands, the first is treated as an action and the remaining as conditions.

If *action* is ' - ', the shell shall reset each *condition* to the default value. If *action* is null ( " " ), the shell shall ignore each specified *condition* if it arises. Otherwise, the argument *action* shall be read and executed by the shell when one of the corresponding conditions arises. The action of [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 [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 the tables of signal names in the [<signal.h>](#) header defined in XBD [Headers](#); for example, HUP, INT, QUIT, TERM. Implementations may permit names with the SIG prefix or ignore case in signal names as an extension. 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 [trap](#) is invoked, the *action* argument shall be processed in a manner equivalent to:

```
eval action
```

Signals that were ignored on entry to a non-interactive 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.

When a subshell is entered, traps that are not being ignored shall be set to the default actions, except in the case of a command substitution containing only a single [trap](#) command, when the traps need not be altered. Implementations may check for this case using only lexical analysis; for example, if ``trap`` and `$( trap -- )` do not alter the traps in the subshell, cases such as assigning `var=trap` and then using `$( $var )` may still alter them. This does not imply that the [trap](#) command cannot be used within the subshell to set new traps.

The [trap](#) command with no operands shall write to standard output a list of commands associated with each condition. If the command is executed in a subshell, the implementation does not perform the optional check described above for a command substitution containing only a single [trap](#) command, and no [trap](#) commands with operands have been executed since entry to the subshell, the list shall contain the commands that were associated with each condition immediately before the subshell environment was entered. Otherwise, the list shall contain the commands currently associated with each condition. The format shall be:

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

The shell shall format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same trapping results. For example:

```
save_traps=$(trap)
```

```
...  
eval "$save_traps"
```

[XSI] ⓘ XSI-conformant systems also allow numeric signal numbers for the conditions corresponding to the following signal names:

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

⏮

The `trap` special built-in shall conform to XBD [Utility Syntax Guidelines](#).

## OPTIONS

None.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## INPUT FILES

None.

## ENVIRONMENT VARIABLES

None.

## ASYNCHRONOUS EVENTS

Default.

## STDOUT

See the DESCRIPTION.

## STDERR

The standard error shall be used only for diagnostic messages.





## OUTPUT FILES

None.

## EXTENDED DESCRIPTION

None.

## EXIT STATUS

If the trap name [\[XSI\]](#)  or number  is invalid, a non-zero exit status shall be returned; otherwise, zero shall be returned. For both interactive and non-interactive shells, invalid signal names [\[XSI\]](#)  or numbers  shall not be considered a syntax error and do not cause the shell to abort.

## CONSEQUENCES OF ERRORS

Default.

---

*The following sections are informative.*

## APPLICATION USAGE

None.

## EXAMPLES

Write out a list of all traps and actions:

```
trap
```

Set a trap so the *logout* utility in the directory referred to by the *HOME* environment variable executes 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
```

## RATIONALE

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 this volume of POSIX.1-2008 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.

Trapping SIGKILL or SIGSTOP is syntactically accepted by some historical implementations, but it has no effect. Portable POSIX applications cannot attempt to trap these signals.



The output format is not historical practice. Since the output of historical [trap](#) commands 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.

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, for example, 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
```

The command:

```
trap 'eval " $cmd"' 0
```

causes the contents of the shell variable *cmd* to be executed as a command when the shell exits. Using:

```
trap '$cmd' 0
```

does not work correctly if *cmd* contains any special characters such as quoting or redirections. Using:

```
trap " $cmd" 0
```

also works (the leading <space> character protects against unlikely cases where *cmd* is a decimal integer or begins with ' - ' ), but it expands the *cmd* variable when the [trap](#) command is executed, not when the exit action is executed.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

XBD [Utility Syntax Guidelines](#), [<signal.h>](#)

## CHANGE HISTORY

### Issue 6

XSI-conforming implementations provide the mapping of signal names to numbers given above (previously this had been marked obsolescent). Other implementations need not provide this optional mapping.

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

Austin Group Interpretation 1003.1-2001 #116 is applied.

POSIX.1-2008, Technical Corrigendum 1, XCU/TC1-2008/0052 [53,268,440], XCU/TC1-2008/0053 [53,268,440], XCU/TC1-2008/0054 [163], XCU/TC1-2008/0055 [163], and XCU/TC1-2008/0056 [163] are applied.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---

## NAME

unset - unset values and attributes of variables and functions

## SYNOPSIS

unset [-fv] *name*...

## DESCRIPTION

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 remove it from the environment. Read-only variables cannot be unset.

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

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

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

The [unset](#) special built-in shall support XBD [Utility Syntax Guidelines](#).

Note that:

VARIABLE=

is not equivalent to an [unset](#) of **VARIABLE**; in the example, **VARIABLE** is set to "". Also, the variables that can be [unset](#) should not be misinterpreted to include the special parameters (see [Special Parameters](#)).

## OPTIONS

See the DESCRIPTION.

## OPERANDS

See the DESCRIPTION.

## STDIN

Not used.

## ***INPUT FILES***

None.

## ***ENVIRONMENT VARIABLES***

None.

## ***ASYNCHRONOUS EVENTS***

Default.

## ***STDOUT***

Not used.

## ***STDERR***

The standard error shall be used only for diagnostic messages.

## ***OUTPUT FILES***

None.

## ***EXTENDED DESCRIPTION***

None.

## ***EXIT STATUS***

- 0 All *name* operands were successfully unset.
- >0 At least one *name* could not be unset.

## ***CONSEQUENCES OF ERRORS***

Default.

---

*The following sections are informative.*

## ***APPLICATION USAGE***

None.

## ***EXAMPLES***

Unset *VISUAL* variable:

```
unset -v VISUAL
```

Unset the functions **foo** and **bar**:

```
unset -f foo bar
```

## RATIONALE

Consideration was given to omitting the **-f** option in favor of an *unfunction* utility, but the standard developers decided to retain historical practice.

The **-v** option was introduced because System V historically used one name space for both variables and functions. When [unset](#) is used without options, System V historically unset either a function or a variable, and there was no confusion about which one was intended. A portable POSIX application can use [unset](#) without an option to unset a variable, but not a function; the **-f** option must be used.

## FUTURE DIRECTIONS

None.

## SEE ALSO

[Special Built-In Utilities](#)

XBD [Utility Syntax Guidelines](#)

## CHANGE HISTORY

### Issue 6

IEEE Std 1003.1-2001/Cor 1-2002, item XCU/TC1/D6/5 is applied so that the reference page sections use terms as described in the Utility Description Defaults ([Utility Description Defaults](#)). No change in behavior is intended.

### Issue 7

SD5-XCU-ERN-97 is applied, updating the SYNOPSIS.

*End of informative text.*

---

[<<< Previous](#)

[Home](#)

[Next >>>](#)

---