The Open Group Base Specifications Issue 6
IEEE Std 1003.1, 2004 Edition

A newer edition of this document exists here

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 IEEE Std 1003.1-2001.

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 IEEE Std 1003.1-2001. 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 Word Expansions.
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 IEEE Std 1003.1-2001:

```
*  ?  [  #  ~  =  %
```

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>s 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 dollar sign, backquote, 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 from a file, from a terminal in the case of an interactive shell, or from a string in the case of *sh* **-c** or *system()*. 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 shall be delimited. If there is no current token, the end-of-input indicator shall be returned as the token.
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 current 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 current 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 (`'\'`, `'"`, or `' )'` 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 quote 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 <newline>, the current token shall be delimited.
8. If the current character is an unquoted <blank>, any token containing the previous character is delimited and the current character shall be discarded.
9. If the previous character was part of a word, the current character shall be appended to that word.
10. 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.
11. 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

[UP XSI] ⌦ The processing of aliases shall be supported on all XSI-conformant systems or if the system supports the User Portability Utilities option (and the rest of this section is not further marked for these options). ⌫

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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.10, 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 IEEE Std 1003.1-2001, 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:

!	**do**	**esac**	**in**
{	**done**	**fi**	**then**
}	**elif**	**for**	**until**
case	**else**	**if**	**while**

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:

[[ ]] function select

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 Word Expansions for a detailed summary of all the stages involved in expanding words.

@
: Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, and where field splitting (see Field Splitting) is performed, each positional parameter shall expand as a separate field, with the provision that the expansion of the first parameter shall still be joined with the beginning part of the original word (assuming that the expanded parameter was embedded within a word), and the expansion of the last parameter shall still be joined with the last part of the original word. If there are no positional parameters, the expansion of `'@'` shall generate zero fields, even when `'@'` is double-quoted.

*
: Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string (see Double-Quotes), it shall expand to a single field with the value of each parameter separated by the first character of the *IFS* variable, or by a <space> if *IFS* is unset. If *IFS* is set to a null string, this is not equivalent to unsetting it; its first character does not exist, so the parameter values are concatenated.

#
: 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 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables and the *exec* function in the System Interfaces volume of IEEE Std 1003.1-2001) 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 XSI] ⊠ The processing of the *ENV* shell variable shall be supported on all XSI-conformant systems or 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

(Input Field Separators.) A string treated as a list of characters that is used for field splitting and to split lines into fields with the *read* command. If *IFS* is not set, the shell shall behave as if the value of *IFS* is <space>, <tab>, and <newline>; see Field Splitting. Implementations may ignore the value of *IFS* in the environment at the time the shell is invoked, treating *IFS* as if it were not set.

LANG

Provide a default value for the internationalization variables that are unset or null. (See the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.2, 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 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, 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>s (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 IEEE Std 1003.1-2001 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 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, Environment Variables, used to effect command interpretation; see Command Search and Execution.

PPID

Set by the shell to the decimal process ID of the process that invoked this 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. This volume of IEEE Std 1003.1-2001 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

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 IEEE Std 1003.1-2001 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 IEEE Std 1003.1-2001 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 IEEE Std 1003.1-2001 specifies the effects of the variable only for systems supporting the User Portability Utilities option.

PWD

Set by the shell to be an absolute pathname of the current working directory, containing no components of type symbolic link, no components that are dot, and no components that are dot-dot when the shell is initialized. 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 either not numeric, the name of one of the special parameters (see Special Parameters), a valid first character of a variable name, a left curly brace (`'{'`) or 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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 4.21, Variable Assignment), multiple tilde-prefixes can be used: at the beginning of the word (that is, following the equal 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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 8.3, 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 IEEE Std 1003.1-2001. If the system does not recognize the login name, the results are undefined.

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 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 name or symbol is not enclosed in braces, the expansion shall use the longest valid name (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.230, Name), whether or not the symbol represented by that name exists.

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, with the exception of '@' ; see Special Parameters.

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. (For example, ${ **foo-bar**} **xyz**} would result in the expansion of **foo** followed by the string **xyz**} if **foo** is set, else the string "barxyz}").

${*parameter*:-*word*}
: **Use Default Values**. If *parameter* is unset or null, the expansion of *word* 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* 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* 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. The following table summarizes the effect of the colon:

	parameter	***parameter***	***parameter***
	Set and Not Null	**Set But Null**	**Unset**
${*parameter***:-***word***}**	substitute *parameter*	substitute *word*	substitute *word*
${*parameter***-***word***}**	substitute *parameter*	substitute null	substitute *word*

${*parameter***:=***word***}**	substitute *parameter*	assign *word*	assign *word*
${*parameter***=***word***}**	substitute *parameter*	substitute null	assign *word*
${*parameter***:?***word***}**	substitute *parameter*	error, exit	error, exit
${*parameter***?***word***}**	substitute *parameter*	substitute null	error, exit
${*parameter***:+***word***}**	substitute *word*	substitute null	substitute null
${*parameter***+***word***}**	substitute *word*	substitute *word*	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.

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. 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.

${*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.
${*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.
${*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*:-*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>s at the end of the substitution. Embedded <newline>s 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.

Within the backquoted style of command substitution, backslash shall retain its literal meaning, except when followed by: '$', '`', or '\' (dollar sign, backquote, backslash). The search for the matching backquote shall be satisfied by the first backquote found without a preceding backslash; 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 backslashes, for example:

```
\`command\`
```

If the command substitution consists of a single subshell, such as:

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

a conforming application shall separate the "$(" and '(' into two tokens (that is, separate them with white space). This is required to avoid any ambiguities with arithmetic expansion.

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, 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, the expansion fails and the shell shall write a 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 to split the results of parameter expansion and command substitution into fields.

1. If the value of *IFS* is a <space>, <tab>, and <newline>, or if it is unset, any sequence of <space>s, <tab>s, or <newline>s 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>s and <tab>s 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: `'\'`, `'"'`, and `''` (backslash, single-quote, 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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.165, 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>s 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 IEEE Std 1003.1-2001 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 lines contained in a shell input file, known as a "here-document", to the input of a command.

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>s 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).

If any character in *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 characters in *word* are 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 symbol is "<<-", all leading <tab>s 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.

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

For a non-interactive shell, an error condition encountered by a special built-in (see Special Built-In Utilities) or other type of utility shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

Error	Special Built-In	Other Utilities
Shell language syntax error	Shall exit	Shall exit
Utility syntax error (option or operand error)	Shall exit	Shall not exit

Redirection error	Shall exit	Shall not exit
Variable assignment error	Shall exit	Shall not exit
Expansion error	Shall exit	Shall exit
Command not found	N/A	May exit
Dot script not found	Shall exit	N/A

An expansion error is one that occurs when the shell expansions defined in Word Expansions 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, the subshell shall (respectively may) exit with a non-zero status, but the script containing the subshell shall not exit because of the error.

In all of the cases shown in the table, an interactive shell shall write a diagnostic message to standard error without exiting.

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 greater than zero.

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 IEEE Std 1003.1-2001). 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. 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 for the processing of special built-in utilities; see Special Built-In Utilities.

If no command name results, variable assignments shall affect the current execution environment. Otherwise, the variable assignments shall be exported for the execution environment of the command and shall not affect the current execution environment (except for special built-ins). If any of the variable assignments attempt to assign a value to a read-only variable, 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 slashes, 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 function known to this 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 1d.
    c. If the command name matches the name of a utility listed in the following table, that utility shall be invoked.

alias	*false*	*jobs*	*read*	*wait*
bg	*fc*	*kill*	*true*	
cd	*fg*	*newgrp*	*umask*	

command *getopts* *pwd* *unalias*

d. Otherwise, the command shall be searched for using the *PATH* environment variable as described in the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 8, 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 *execve*() function as defined in the System Interfaces volume of IEEE Std 1003.1-2001 with the *path* argument set to the pathname resulting from the search, *arg*0 set to the command name, and the remaining arguments set to the operands, if any.

If the *execve*() function fails due to an error equivalent to the [ENOEXEC] error defined in the System Interfaces volume of IEEE Std 1003.1-2001, 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.

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 *execve*() function defined in the System Interfaces volume of IEEE Std 1003.1-2001 with the *path* and *arg*0 arguments set to the command name, and the remaining arguments set to the operands, if any.

If the *execve*() 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.

2.9.2 Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator `'|'`. The standard output of all but the last command shall be connected to the standard input of the next command.

The format for a pipeline is:

[!] *command1* **[** | *command2* **...]**

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 reserved word **!** does not precede the pipeline, 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 '`&`' and optionally terminated by '`;`', '`&`', or <newline>.

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 '`;`' or <newline> terminator shall cause the preceding AND-OR list to be executed sequentially; an '`&`' 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>s, that can be preceded or followed by an arbitrary number of <newline>s.

The following sections are informative.

Examples

The following is an example that illustrates <newline>s 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 ampersand ('`&`'), 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* **&** **...]**

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**. If it is an interactive shell, this need not happen. 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 ampersand, 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 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.

{ *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 *pattern*s 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-listthen
    compound-list[elif compound-listthen
    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-1do
    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-1do
    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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.230, Name). 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 Word Expansions 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 and variable-assignment properties described for special built-in utilities 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* 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. A <newline> shall be returned as the token identifier **NEWLINE**.
2. If the token is an operator, the token identifier for that operator shall result.
3. If the string consists solely of digits and the delimiter character is one of `'<'` or `'>'`, the token identifier **IO_NUMBER** shall be returned.
4. 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**, 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 quote marks 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>s 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 the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.230, 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>s 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 the equal sign character:

- If it begins with '=', the token **WORD** shall be returned.
- If all the characters preceding '=' form a valid name (see the Base Definitions volume of IEEE Std 1003.1-2001, Section 3.230, Name), the token **ASSIGNMENT_WORD** shall be returned. (Quoted characters cannot participate in forming a valid name.)
- Otherwise, it is unspecified whether it is **ASSIGNMENT_WORD** or **WORD** that is returned.

Assignment to the **NAME** 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 mentioned above. */


%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  complete_command
%%
complete_command : list separator
                 | 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    :              term
                 | newline_list term
                 |              term separator
                 | newline_list term separator
                 ;
term             : term separator and_or
                 |                and_or
                 ;
for_clause       : For name linebreak                            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 linebreak
                | '(' pattern ')'               linebreak
                | '(' pattern ')' compound_list linebreak
                ;
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 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

When a command is in an asynchronous list, the shell shall prevent SIGQUIT and SIGINT signals from the keyboard from interrupting the command. Otherwise, signals shall have the values inherited by the shell from its parent (see also the *trap* special built-in).

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*
- 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 IEEE Std 1003.1-2001 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 set by that shell environment shall be set to the default values. Changes made to the subshell environment shall not affect the shell environment. Command substitution, commands that are grouped with parentheses, and asynchronous lists shall be executed in a subshell environment. Additionally, each command of a 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 the Base Definitions volume of IEEE Std 1003.1-2001, Chapter 9, 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 include backslash escape processing.

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.

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.

`[`
: The open bracket shall introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in the Base Definitions volume of IEEE Std 1003.1-2001, Section 9.3.5, RE Bracket Expression shall also apply to the pattern 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. A bracket expression starting with an unquoted circumflex character produces unspecified results.

When pattern matching is used where shell quote removal is not performed (such as in the argument to the *find* - *name* primary when *find* is being called using one of the *exec* functions as defined in the System Interfaces volume of IEEE Std 1003.1-2001, 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 asterisks is a valid pattern. In such patterns, each asterisk shall match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

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 slashes in the pattern; it shall neither be matched by the asterisk or question-mark special characters nor by a bracket expression. Slashes 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 open square bracket character before a corresponding closing 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 the pattern contains an invalid bracket expression or 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. A syntax error in a special built-in utility may cause a shell executing that utility to abort, while a syntax error in a regular built-in utility shall not cause a shell executing that utility to abort. (See Consequences of Shell Errors for the consequences of errors on interactive and non-interactive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value shall be non-zero.

2. Variable assignments specified with special built-in utilities 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 IEEE Std 1003.1-2001.

Some of the special built-ins are described as conforming to the Base Definitions volume of IEEE Std 1003.1-2001, Section 12.2, 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.