IEEE P1003.2 Draft 11.2 – September 1991

Copyright © 1991 by the
Institute of Electrical and Electronics Engineers, Inc.
345 East 47th Street
New York, NY 10017, USA
All rights reserved as an unpublished work.

This is an unapproved and unpublished IEEE Standards Draft, subject to change. The publication, distribution, or copying of this draft, as well as all derivative works based on this draft, is expressly prohibited except as set forth below.

Permission is hereby granted for IEEE Standards Committee participants to reproduce this document for purposes of IEEE standardization activities only, and subject to the restrictions contained herein.

Permission is hereby also granted for member bodies and technical committees of ISO and IEC to reproduce this document for purposes of developing a national position, subject to the restrictions contained herein.

Permission is hereby also granted to the preceding entities to make limited copies of this document in an electronic form only for the stated activities.

The following restrictions apply to reproducing or transmitting the document in any form: 1) all copies or portions thereof must identify the document's IEEE project number and draft number, and must be accompanied by this entire notice in a prominent location; 2) no portion of this document may be redistributed in any modified or abridged form without the prior approval of the IEEE Standards Department.

Other entities seeking permission to reproduce this document, or any portion thereof, for standardization or other activities, must contact the IEEE Standards Department for the appropriate license.

Use of information contained in this unapproved draft is at your own risk.

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]

3.9 Shell Commands

This clause 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 description of the command language, consult the grammar in 3.10.

A *command* is one of the following:

— *simple command* (see 3.9.1)

— *pipeline* (see 3.9.2)

— *list* or *compound-list* (see 3.9.3)

— *compound command* (see 3.9.4)

— *function definition* (see 3.9.5).

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command. There is no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, etc.).

3.9.0.1 Shell Commands Rationale. *(This subclause is not a part of P1003.2)*

A description of an ‘‘empty command’’ was removed from an earlier draft because it is only relevant in the cases of `sh -c ""`, `system("")`, or an empty shell-script file (such as the implementation of `true` on some historical systems). Since it is no longer mentioned in POSIX.2, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications will not construct empty commands. (However, note that `sh` does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra `<newline>`s or semicolons, such as

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

3.9.1 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 (i.e., when any conditional construct such as an AND-OR list or a `case` statement has not bypassed the

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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 3.10.2 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 shall be the arguments for the command.

(3) Redirections shall be performed as described in 3.7.

(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 3.14.

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 3.8.1 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 `exec` (3.14.6)]. 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 3.8.1 for the consequences of these failures on interactive and noninteractive shells.

If there is a command name, execution shall continue as described in 3.9.1.1. 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.

3.9.1.0.1 Simple Commands Rationale. *(This subclause is not a part of P1003.2)*

The enumerated list is used only when the command is actually going to be executed. For example, in:

```
true || $foo *
```

no expansions are performed.

The following example illustrates both how a variable assignment without a command name affects the current execution environment, and how an assignment with a command name only affects the execution environment of the command.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
$ x=red
$ echo $x
red
$ export x
$ sh –c 'echo $x'
red
$ x=blue sh –c 'echo $x'
blue
$ echo $x
red
```

This next example illustrates that redirections without a command name are still performed.

```
$ ls foo
ls: foo: no such file or directory
$ > foo
$ ls foo
foo
```

Historical practice is for a command without a command name, but that includes a command substitution, to have an exit status of the last command substitution that the shell performed and some historical scripts rely on this. For example:

```
if      x=$(command )
then    ...
fi
```

An example of redirections without a command name being performed in a subshell shows that the here-document does not disrupt the standard input of the `while` loop:

```
IFS=:
while   read a b
do      echo $a
        <<-eof
        Hello
        eof
done </etc/passwd
```

Some examples of commands without command names in AND/OR lists:

```
> foo || {
        echo "error: foo cannot be created" >&2
        exit 1
}

# set saved if /vmunix.save exists
test -f /vmunix.save && saved=1
```

Command substitution and redirections without command names both occur in subshells, but they are not the same ones. For example, in:

```
exec 3> file
var=$(echo foo >&3) 3>&1
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

it is unspecified whether `foo` will be echoed to the file or to standard output.

3.9.1.1 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 shell, the function shall be invoked as described in 3.9.5. [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 (1)(d).]

(c) If the command name matches the name of a utility listed in Table 2-2 (see 2.3), that utility shall be invoked.

(d) Otherwise, the command shall be searched for using the **PATH** environment variable as described in 2.6:

[1] 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 shall execute the utility in a separate utility environment (see 3.12) with actions equivalent to calling the POSIX.1 {8} *execve*() function with the *path* argument set to the pathname resulting from the search, *arg0* 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 POSIX.1 {8} error [ENOEXEC], the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message, and 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 startup 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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

repeat the search to find the new location for the utility, if any.

[2] 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 does contain slashes, the shell shall execute the utility in a separate utility environment with actions equivalent to calling the POSIX.1 {8} *execve*() function with the *path* and *arg0* 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 POSIX.1 {8} error [ENOEXEC], the shell shall execute a command equivalent to having a shell invoked with the command name as its first operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message, and return an exit status of 126.

3.9.1.1.1 Command Search and Execution Rationale. *(This subclause is not a part of P1003.2)*

This description requires that the shell can execute shell scripts directly, even if the underlying system does not support the common #! interpreter convention. That is, if file `foo` contains shell commands and is executable, the following will execute `foo`:

```
./foo
```

The command search shown here does not match all historical implementations. A more typical sequence has been:

— Any built-in, special or regular.

— Functions.

— Path search for executable files.

But there are problems with this sequence. Since the programmer has no idea in advance which utilities might have been built into the shell, a function cannot be used to portably override a utility of the same name. (For example, a function named `cd` cannot be written for many historical systems.) Furthermore, the **PATH** variable is partially ineffective in this case and only a pathname with a slash can be used to ensure a specific executable file is invoked.

The sequence selected for POSIX.2 acknowledges that special built-ins cannot be overridden, but gives the programmer full control over which versions of other utilities are executed. It provides a means of suppressing function lookup (via the `command` utility; see 4.12) for the user's own functions and ensures that any regular built-ins or functions provided by the implementation are under the control of the path search. The mechanisms for associating built-ins or functions with executable files in the path are not specified by POSIX.2, but the wording requires that if either is implemented, the application will not be able to distinguish a function or built-in from an executable (other than in terms of performance,

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

presumably). The implementation must ensure that all effects specified by POSIX. 2 resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, etc.) are identical to those resulting from the invocation of an executable file.

Example : Consider three versions of the `ls` utility:

— The application includes a shell function named `ls`.

— The user writes her own utility named `ls` and puts it in `/hsa/bin`.

— The example implementation provides `ls` as a regular shell built-in that will be invoked (either by the shell or directly by *exec*) when the path search reaches the directory `/posix/bin`.

If `PATH=/posix/bin`, various invocations yield different versions of `ls`:

Invocation	Version of `ls`
`ls` (from within application script)	(1) function
`command ls` (from within application script)	(3) built-in
`ls` (from within makefile called by application)	(3) built-in
`system("ls")`	(3) built-in
`PATH="/hsa/bin:$PATH" ls`	(2) user's version

After the *execve* () failure described, the shell normally executes the file as a shell script. Some implementations, however, attempt to detect whether the file is actually a script and not an executable from some other architecture. The method used by the KornShell is allowed by the text that indicates nontext files may be bypassed.

3.9.2 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 3.7).

If the pipeline is not in the background (see 3.9.3.1), the shell shall wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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 is 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 is zero.

3.9.2.1 Pipelines Rationale. *(This subclause is not a part of P1003.2)*

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

```
$ command1 2>&1 | command2
```

sends both the standard output and standard error of `command1` to the standard input of `command2`.

The reserved word ! was added to allow more flexible testing using AND and OR lists.

It was suggested that it would be better to return a nonzero value if any command in the pipeline terminates with nonzero status (perhaps the bitwise OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause application breakage if changed. An example of historical (and POSIX. 2) behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5
$ echo $?
0
```

3.9.3 Lists

An *AND-OR-list* is a sequence of one or more pipelines separated by the operators

```
&&     ||
```

A *list* is a sequence of one or more AND-OR-lists separated by the operators

```
;     &
```

and optionally terminated by

```
;     &     <newline>
```

The operators && and || shall have equal precedence and shall be evaluated from beginning to end.

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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

The term *compound-list* is derived from the grammar in 3.10; 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.

3.9.3.0.1 Lists Rationale. *(This subclause is not a part of P1003.2)*

The equal precedence of && and || is historical practice. The developers of the standard evaluated the model used more frequently in high level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but could not in the end allow existing scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the { } or ( ) groupings that are required historically. There are some disadvantages to these groupings:

— The ( ) can be expensive, as they spawn other processes on some systems. This performance concern is primarily an implementation issue.

— The { } braces are not operators (they are reserved words) and require a trailing space after each {, and a semicolon before each }. Most programmers (and certainly interactive users) have avoided braces as grouping constructs because of the irritating syntax required. Braces were not changed to operators because that would generate compatibility issues even greater than the precedence question; braces appear outside the context of a keyword in many shell scripts.

An example reiterates the precedence of the lists as they associate from beginning to end. Both of the following commands write solely bar to standard output:

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

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

```
while
        # a couple of newlines

        # a list
        date && who || ls; cat file
        # a couple of newlines

        # another list
        wc file > output & true

do
        # 2 lists
        ls
        cat file
done
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.9.3.1 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 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 3.12. This process ID shall remain known until:

— The command terminates and the application waits for the process ID, or

— Another asynchronous list is invoked before `$!` (corresponding to the previous asynchronous list) is expanded in the current execution environment.

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

Exit Status

The exit status of an asynchronous list shall be zero.

3.9.3.1.1 Asynchronous Lists Rationale. *(This subclause is not a part of P1003.2)*

The grammar treats a construct such as

```
foo & bar & bam &
```

as one ‘‘asynchronous list,’’ but since the status of each element is tracked by the shell, the term ‘‘element of an asynchronous list’’ was introduced to identify just one of the `foo`, `bar`, `bam` portions of the overall list.

Unless the implementation has an internal limit, such as {CHILD_MAX}, on the retained process IDs, it would require unbounded memory for the following example:

```
while true
do      foo & echo $!
done
```

The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in 3.11.

Since the connection of the input to the equivalent of `/dev/null` is considered to occur before redirections, the following script would produce no output:

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

```
exec < /etc/passwd
cat <&0 &
wait
```

3.9.3.2 Sequential Lists

Commands that are separated by a semicolon (;) shall be executed sequentially.

The format for executing commands sequentially is:

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.

3.9.3.3 AND Lists

The control operator && shall denote an AND list. The format is:

command1 [&& *command2*] ...

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

3.9.3.4 OR Lists

The control operator || shall denote an OR List. The format is:

command1 [|| *command2*] ...

First, *command1* is executed. If its exit status is nonzero, *command2* is 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.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.9.4 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.

3.9.4.1 Grouping Commands

The format for grouping commands is as follows:

`(`*compound-list* `)`	Execute *compound-list* in a subshell environment; see 3.12. 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.

Exit Status

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

3.9.4.1.1 Grouping Commands Rationale. *(This subclause is not a part of P1003.2)*

The semicolon shown in `{` *compound-list* `;}` is an example of a control operator delimiting the `}` reserved word. Other delimiters are possible, as shown in 3.10; `<newline>` is frequently used.

A proposal was made to use the <do-done> construct in all cases where command grouping performed in the current process environment is performed, identifying it as a construct for the grouping commands, as well as for shell functions. This was not included because the shell already has a grouping construct for this purpose (`{ }`), and changing it would have been counter-productive.

3.9.4.2 `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
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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 ...
```

is 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.

3.9.4.2.1 `for` Loop Rationale. *(This subclause is not a part of P1003.2)*

The format is shown with generous usage of `<newline>`s. See the grammar in 3.10 for a precise description of where `<newline>`s and semicolons can be interchanged.

Some historical implementations support `{` and `}` as substitutes for `do` and `done`. The working group chose to omit them, even as an obsolescent feature. (Note that these substitutes were only for the `for` command; the `while` and `until` commands could not use them historically, because they are followed by compound-lists that may contain `{...}` grouping commands themselves

The reserved word pair `do` ... `done` was selected rather than `do` ... `od` (which would have matched the spirit of `if` ... `fi` and `case` ... `esac`) because `od` is a commonly-used utility name and this would have been an unacceptable choice.

3.9.4.3 `case` Conditional Construct

The conditional construct `case` shall execute the *compound-list* corresponding to the first one of several *patterns* (see 3.13) that is matched by the string resulting from the tilde expansion, parameter expansion, command substitution, and 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* are 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 is 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 ;;
        [(]pattern2 |pattern3 ) compound-list ;;
        ...
esac
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

Each pattern in a pattern list shall be expanded and compared against the expansion of *word* . After the first match, no more patterns shall be expanded, and the *compound-list* shall be executed. The order of expansion and comparing of patterns in a multiple pattern list is unspecified.

Exit Status

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

3.9.4.3.1 `case` Conditional Construct Rationale.

(This subclause is not a part of P1003.2)

An optional open-parenthesis before *pattern* was added to allow numerous historical KornShell scripts to conform. At one time, using the leading parenthesis was required if the `case` statement were to be embedded within a `$( )` command substitution; this is no longer the case with the POSIX shell. Nevertheless, many existing scripts use the open-parenthesis, if only because it makes matching-parenthesis searching easier in `vi` and other editors. This is a relatively simple implementation change that is fully upward compatible for all scripts.

Consideration was given to requiring `break` inside the *compound-list* to prevent falling through to the next pattern action list. This was rejected as being nonexisting practice. An interesting undocumented feature of the KornShell is that using `;&` instead of `;;` as a terminator causes the exact opposite behavior—the flow of control continues with the next *compound-list* .

The pattern `"*"`, given as the last pattern in a `case` construct, is equivalent to the default case in a C-language `switch` statement

The grammar shows that reserved words can be used as patterns, even if one is the first word on a line. Obviously, the reserved word `esac` cannot be used in this manner.

3.9.4.4 `if` Conditional Construct

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

The format for the `if` construct is as follows.

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

3.9.4.4.1 `if` Conditional Construct Rationale.

(This subclause is not a part of P1003.2)

The precise format for the command syntax is described in 3.10.

3.9.4.5 `while` Loop

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

The format of the `while` loop is as follows

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

The *compound-list-1* shall be executed, and if it has a nonzero 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.

3.9.4.5.1 `while` Loop Rationale.

(This subclause is not a part of P1003.2)

The precise format for the command syntax is described in 3.10.

3.9.4.6 `until` Loop

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

The format of the `until` loop is as follows

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

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

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

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.

3.9.4.6.1 `until` Loop Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.5 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* ; it shall be a name (see 3.1.5). An implementation may allow other characters in a function name as an extension. The implementation shall maintain separate namespaces for functions and variables.

The argument *compound-command* represents a compound command, as described in 3.9.4.

When the function is declared, none of the expansions in 3.6 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 3.8.1 for the consequences of failures of these operations on interactive and noninteractive 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 3.14.

The *compound-command* shall be executed whenever the function name is specified as the name of a simple command (see 3.9.1.1). The operands to the command temporarily shall become the positional parameters during the execution of the *compound-command* ; the special parameter # shall also 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 shall complete and execution shall resume with the next command after the function call.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

1895 **Exit Status**

1896 The exit status of a function definition shall be zero if the function was declared
1897 successfully; otherwise, it shall be greater than zero. The exit status of a function
1898 invocation shall be the exit status of the last command executed by the function.

1899 **3.9.5.1 Function Definition Command Rationale** *(This subclause is not a part of*
1900 *P1003.2)*

1901 The description of functions in Draft 8 was based on the notion that functions
1902 should behave like miniature shell scripts; that is, except for sharing variables,
1903 most elements of an execution environment should behave as if it were a new exe-
1904 cution environment, and changes to these should be local to the function. For
1905 example, traps and options should be reset on entry to the function, and any
1906 changes to them don't affect the traps or options of the caller. There were
1907 numerous objections to this basic idea, and the opponents asserted that functions
1908 were intended to be a convenient mechanism for grouping commonly executed
1909 commands that were to be executed in the current execution environment, similar
1910 to the execution of the `dot` special built-in.

1911 Opponents also pointed out that the functions described in Draft 8 did not scope
1912 everything a new shell script would anyway, such as the current working direc-
1913 tory, or umask, but instead picked a few select properties. The basic argument
1914 was that if one wanted scoping of the execution environment, the mechanism
1915 already exists: put the commands in a new shell script and call it. All traditional
1916 shells that implemented functions, other than the KornShell, have implemented
1917 functions that operate in the current execution environment. Because of this,
1918 Draft 9 removed any local scoping of traps or options. Local variables within a
1919 function were considered and included in Draft 9 (controlled by the special built-
1920 in `local`), but were removed because they do not fit the simple model developed
1921 for the scoping of functions and there was some opposition to adding yet another
1922 new special built-in from outside existing practice. Implementations should
1923 reserve the identifier `local` (as well as `typeset`, as used in the KornShell) in
1924 case this local variable mechanism is adopted in a future version of POSIX. 2.

1925 A separate issue from the execution environment of a function is the availability
1926 of that function to child shells. A few objectors, including the author of the origi-
1927 nal Version 7 UNIX system shell, maintained that just as a variable can be shared
1928 with child shells by exporting it, so should a function—and so this capability has
1929 been added to the standard. In previous drafts, the `export` command therefore
1930 had a `-f` flag for exporting functions. Functions that were exported were to be
1931 put into the environment as *name* `( )`=*value* pairs, and upon invocation, the shell
1932 would scan the environment for these, and automatically define these functions.
1933 This facility received a lot of balloting opposition and was removed from Draft 11.
1934 Some of the arguments against exportable functions were:

1935 — There was little existing practice. The Ninth Edition shell provided them,
1936 but there was controversy over how well it worked.

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

— There are numerous security problems associated with functions appearing in a script's environment and overriding standard utilities or the application's own utilities.

— There was controversy over requiring `make` to import functions, where it has historically used an *exec* function for many of its command line executions.

— Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different than variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this insight might be lost on the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in POSIX. 2 match those in System V. The KornShell has two methods of defining functions:

`function` *fname* `{` *compound-list* `}`

and

fname `() {` *compound-list* `}`

The latter uses the same definition as POSIX. 2, but differs in semantics, as described previously. A future edition of the KornShell is planned to align the latter syntax with POSIX and keep the former as-is.

The name space for functions is limited to that of a *name* because of historical practice. Complications in defining the syntactic rules for the function definition command and in dealing with known extensions such as the KornShell's `@()` prevented the name space from being widened to a *word* , as requested by some balloters. Using functions to support synonyms such as the C-shell's `!!` and `%` is thus disallowed to portable applications, but acceptable as an extension. For interactive users, the aliasing facilities in the UPE should be adequate for this purpose. It is recognized that the name space for utilities in the file system is wider than that currently supported for functions, if the portable filename character set guidelines are ignored, but it did not seem useful to mandate extensions in systems for so little benefit to portable applications.

The `()` in the function definition command consists of two operators. Therefore, intermixing `<blank>`s with the *fname* , `(`, and `)` is allowed, but unnecessary.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls -l
#
[ X$i = Xyes ] && foo() {
        ls -l
}
```

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.

3.10 Shell Grammar

The following grammar describes the Shell Command Language. Any discrepancies found between this grammar and the preceding description shall be resolved in favor of this clause.

3.10.1 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 are used to determine what a ''token'' that is subject to parsing at the token level is. The rules for token recognition in 3.3 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` shall result.

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.

3.10.2 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. 1
Also, if the parser is in any state where only a reserved word could be the 1
next correct token, proceed as above. 1

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 will not be recognized except in certain positions in the input, such as after a `<newline>` or

Copyright © 1991 IEEE. All rights reserved.
This is an unapproved IEEE Standards Draft, subject to change.