-- --

IEEE P1003.2 Draft 11.2 – September 1991

3.1 Shell Definitions

The following terms are used in Section 3. Because they are specific to the shell, they do not appear in 2.2.2.

3.1.1 control operator: A token that performs a control function.

It is one of the following symbols:

&	)	<newline>
&&	;	|
(	;;	||

The end-of-input indicator used internally by the shell is also considered a control operator. See 3.3.

On some systems, the symbol `((` is a control operator; its use produces 1
unspecified results.

3.1.2 expand: When not qualified, the act of applying all the expansions described in 3.6.

3.1.3 field: A unit of text that is the result of parameter expansion (3.6.2), arithmetic expansion (3.6.4), command substitution (3.6.3), or field splitting (3.6.5).

During command processing (see 3.9.1), the resulting fields are used as the command name and its arguments.

3.1.4 interactive shell: A processing mode of the shell that is suitable for direct user interaction.

The behavior in this mode is not defined by this standard.

NOTE: The preceding sentence is expected to change following the eventual approval of the UPE supplement.

3.1.5 name: A word consisting solely of underscores, digits, and alphabetics from the portable character set (see 2.4).

The first character of a name shall not be a digit.

3.1.6 operator: Either a control operator or a redirection operator.

3.1.7 parameter: An entity that stores values.

There are three types of parameters: variables (named parameters), positional parameters, and special parameters. Parameter expansion is accomplished by introducing a parameter with the `$` character. See 3.5.

3.1.8 positional parameter: A parameter denoted by a single digit or one or more digits in curly braces.

See 3.5.1.

3.1.9 redirection: A method of associating files with the input/output of commands.

See 3.7.

3.1.10 redirection operator: A token that performs a redirection function.

It is one of the following symbols:

< > >| << >> <& >& <<- <>

3.1.11 special parameter: A parameter named by a single character from the following list:

* @ # ? ! – $ 0

See 3.5.2.

3.1.12 subshell: A shell execution environment, distinguished from the main or current shell execution environment by the attributes described in 3.12.

3.1.13 token: A sequence of characters that the shell considers as a single unit when reading input, according to the rules in 3.3.

A token is either an operator or a word.

3.1.14 variable: A named parameter. See 3.5.

3.1.15 variable assignment [assignment]: A word consisting of the following parts

varname =*value*

When used in a context where assignment is defined to occur (see 3.9.1) and at no other time, the *value* (representing a word or field) shall be assigned as the value of the variable denoted by *varname* . The *varname* and *value* parts meet the requirements for a name and a word, respectively, except that they are delimited by the embedded unquoted equals-sign in addition to the delimiting described in 3.3. In all cases, the variable shall be created if it did not already exist. If *value* is not specified, the variable shall be given a null value.

An alternative form of variable assignment:

symbol =*value*

(where *symbol* is a valid word delimited by an equals-sign, but not a valid name) produces unspecified results.

3.1.16 word: A token other than an operator.

In some cases a word is also a portion of a word token: in the various forms of parameter expansion (3.6.2), such as ${*name* –*word* }, and variable assignment, such as *name*=*word* , the word is the portion of the token depicted by *word* . The concept of a word is no longer applicable following word expansions—only fields remain; see 3.6.

3.1.17 Shell Definitions Rationale. *(This subclause is not a part of P1003.2)*

The *word* =*word* form of variable assignment was included, producing unspecified results, to allow the KornShell *name* [*expression*]=*value* syntax to conform.

The `((` symbol is a control operator in the KornShell, used for an alternative syn- 1
tax of an arithmetic expression command. A strictly conforming POSIX.2 application cannot use `((` as a single token [with the obvious exception of the `$((` form described in POSIX.2]. The decision to require this is based solely on the pragmatic knowledge that there are many more historical shell scripts using the KornShell syntax than there might be using nested subshells, such as

`((foo))` or `((foo);(bar))`

The latter example should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. Thus, in most cases, while a few scripts will no longer be strictly portable, the chances of breaking existing scripts is even smaller.

There are no explicit limits in this standard on the sizes of names, words, lines, or 1
other objects. However, other implicit limits do apply: shell script lines produced 1
by many of the standard utilities cannot exceed {LINE_MAX} and the sum of 1
exported variables comes under the {ARG_MAX} limit. Historical shells dynami- 1
cally allocate memory for names and words and parse incoming lines a byte at a 1
time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical prac- 1
tice such as makefiles, where `make` removes the `<newline>`s associated with the 1
commands for a target and presents the shell with one very long line. The text in 1
2.11.5.2 does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

3.2 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 3.7.4).

The following characters shall be quoted 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 the standard:

* ? [# ~ = %

The various quoting mechanisms are the escape character, single-quotes, and double-quotes. The here-document represents another form of quoting; see 3.7.4.