IEEE P1003.2 Draft 11.2 – September 1991

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 syntax 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 other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a byte at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice such as makefiles, where `make` removes the `<newline>`s associated with the commands for a target and presents the shell with one very long line. The text in 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.

3.2.1 Escape Character (Backslash)

A backslash that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the backslash, the shell shall interpret this as line continuation. The backslash and <newline> shall be removed before splitting the input into tokens.

3.2.2 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.

3.2.3 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 3.6.2), a form of command substitution (see 3.6.3), and arithmetic expansion (see 3.6.4).

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 3.3 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 3.6.2 shall be used to determine the matching }.

‘ The backquote shall retain its special meaning introducing the other form of command substitution (see 3.6.3). 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- 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 3.2.1) only when followed by one of the characters:

`$` `` ` `` `"` `\` `<newline>`

A double-quote shall be preceded by a backslash to be included within double-quotes. The parameter @ has special meaning inside double-quotes and is described in 3.5.2.

3.2.4 Quotes Rationale. *(This subclause is not a part of P1003.2)*

A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded quote can be created by writing, for example, `'a'\''b'`, which yields `a'b`. (See 3.6.5 for a better understanding of how portions of words are either split into fields or remain concatenated.) A single token can be made up of concatenated partial strings containing all three kinds of quoting/escaping, thus permitting any combination of characters.

The escaped `<newline>` used for line continuation is removed entirely from the input and is not replaced by any white space. Therefore, it cannot serve as a token separator.

In double-quoting, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unmodified and the character immediately following it is also left unmodified. Thus, for example:

`"\$"` ⇒ `$`

`"\a"` ⇒ `\a`

It would be desirable to include the statement ‘‘The characters from an enclosed `${` to the matching `}` shall not be affected by the double-quotes,’’ similar to the one for `$( )`. However, historical practice in the System V shell prevents this. The requirement that double-quotes be matched inside `${...}` within double-quotes and the rule for finding the matching `}` in 3.6.2 eliminate several subtle inconsistencies in expansion for historical shells in rare cases; for example,

```
"${foo-bar"}
```

yields `bar` when `foo` is not defined, and is an invalid substitution when `foo` is defined, in many historical shells. The differences in processing the `"${...}"` form have led to inconsistencies between the historical System V, BSD, and KornShells, and the text in POSIX.2 is an attempt to converge them without breaking many applications. A consequence of the new rule is that single-quotes cannot be used to quote the `}` within `"${...}"`; for example

```
unset bar
foo="${bar-'}'}"
```

is invalid because the `"${...}"` substitution contains an unpaired unescaped single-quote. The backslash can be used to escape the `}` in this example to achieve the desired result:

```
unset bar
foo="${bar-\}}"
```

The only alternative to this compromise between shells would be to make the behavior unspecified whenever the literal characters ', {, }, and " appear within `${...}`. To write a portable script that uses these values, a user would have to assign variables, say,

```
squote=\' dquote=\" lbrace='{' rbrace='}'
${foo-$squote$rbrace$squote}
```

rather than

```
${foo-"'}'"}
```

Some systems have allowed the end of the word to terminate the backquoted command substitution, such as in

```
"`echo hello"
```

This usage is undefined in POSIX.2, where the matching backquote is required. The other undefined usage can be illustrated by the example:

```
sh -c '` echo "foo`'
```

The description of the recursive actions involving command substitution can be illustrated with an example. Upon recognizing the introduction of command substitution, the shell must parse input (in a new context), gathering the ‘‘source’’ for the command substitution until an unbalanced `)` or `` ` `` is located. For example, in the following

```
echo "$(date; echo "
        one" )"
```

the double-quote following the `echo` does not terminate the first double-quote; it is part of the command substitution ‘‘script.’’ Similarly, in

```
echo "$(echo *)"
```

the asterisk is not quoted since it is inside command substitution; however,

```
echo "$(echo "*")"
```

is quoted (and represents the asterisk character itself).

3.3 Token Recognition

The shell reads 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 1
input lines can be of unlimited length. These lines are parsed using two major 1
modes: ordinary token recognition and processing of here-documents.

When an `io_here` token has been recognized by the grammar (see 3.10), one or more of the immediately subsequent lines form the body of one or more here-