IEEE P1003.2 Draft 11.2 – September 1991

LC_COLLATE This variable shall determine the behavior of range expressions, equivalence classes, and multicharacter collating elements within pattern matching.

LC_CTYPE This variable shall determine the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters), which characters are defined as letters (character class `alpha`), and the behavior of character classes within pattern matching.

LC_MESSAGES This variable shall determine the language in which messages should be written.

PATH This variable represents a string formatted as described in 2.6, used to effect command interpretation. See 3.9.1.1.

3.5.3.1 Variables Rationale. *(This subclause is not a part of P1003.2)*

A description of **PWD** (which is automatically set by the KornShell whenever the current working directory changes) was omitted because its functionality is easily reproduced using `$(pwd)`.

See the discussion of **IFS** in 3.6.5.1.

Other common environment variables used by historical shells are not specified by this standard, but they should be reserved for the historical uses. For interactive use, other shell variables are expected to be introduced by the UPE (and this rationale will be updated accordingly): **ENV**, **FCEDIT**, **HISTFILE**, **HISTSIZE**, **LINENO**, **PPID**, **PS 1**, **PS 2**, **PS 4**.

Tilde expansion for components of the **PATH** in an assignment such as:

```
PATH=~hlj/bin:~dwc/bin:$PATH
```

is a feature of some historical shells and is allowed by the wording of 3.6.1. Note that the tildes are expanded during the assignment to **PATH**, not when **PATH** is accessed during command search.

3.6 Word Expansions

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

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 is described in 3.5.2.

The order of word expansion shall be as follows:

(1) Tilde Expansion (see 3.6.1), Parameter Expansion (see 3.6.2), Command Substitution (see 3.6.3), and Arithmetic Expansion (see 3.6.4) shall be performed, beginning to end. [See item (5) in 3.3.] 1

(2) Field Splitting (see 3.6.5) shall be performed on fields generated by step (1) unless **IFS** is null.

(3) Pathname Expansion (see 3.6.6) shall be performed, unless `set -f` is in effect.

(4) Quote Removal (see 3.6.7) shall always be performed last.

The expansions described in this clause 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- 1 quote characters. 1

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 3.5.2), a valid first character of a variable name, a left curly brace (`{`), or a left parenthesis, the result is unspecified.

3.6.0.1 Word Expansions Rationale. *(This subclause is not a part of P1003.2)*

IFS is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behavior. All useful applications of **IFS** use it for parsing input of the `read` utility and for splitting the results of parameter and command substitution. New versions of the shell have fixed this bug, and POSIX. 2 requires the corrected behavior.

The rule concerning expansion to a single field requires that if `foo=abc` and `bar=def`, that

```
"$foo""$bar"
```

expands to the single field

```
abcdef
```

The rule concerning empty fields can be illustrated by:

```
$ unset foo
$ set $foo bar '' xyz "$foo" abc
$ for i
> do
>       echo "-$i-"
> done
-bar-
--
-xyz-
--
-abc-
```

Step (1) indicates that Tilde Expansion, Parameter Expansion, Command Substi- 1
tution, and Arithmetic Expansion are all processed simultaneously as they are
scanned. For example, the following is valid arithmetic:

```
x=1
echo $(( $(echo 3)+$x ))
```

An earlier draft stated that Tilde Expansion preceded the other steps, but this is 1
not the case in known historical implementations; if it were, and a referenced 1
home directory contained a $ character, expansions would result within the direc- 1
tory name. 1

3.6.1 Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, 2
followed by all of the characters preceding the first unquoted slash in the word, or 2
all the characters in the word if there is no slash. In an assignment (see 3.1.15), 2
multiple tilde prefixes can be used: at the beginning of the word (i.e., following 2
the equals-sign of the assignment) and/or following any unquoted colon. A tilde 2
prefix in an assignment is terminated by the first unquoted colon or slash. If none 2
of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix 1
following the tilde shall be treated as a possible login name from the user data-
base (see POSIX.1 {8} Section 9). A portable login name cannot contain characters 2
outside the set given in the description of the **LOGNAME** environment variable in 2
POSIX.1 {8}. If the login name is null (i.e., the tilde-prefix contains only the tilde),
the tilde-prefix shall be 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 home directory associated with the login name obtained using
the equivalent of the POSIX.1 {8} *getpwnam*() function. If the system does not 1
recognize the login name, the results are undefined.

3.6.1.1 Tilde Expansion Rationale. *(This subclause is not a part of P1003.2)*

The text about quoting of the word indicates that `\~hlj/`, `~h\lj/`, `~"hlj"/`, `~hlj\/`, and `~hlj/` are not equivalent: only the last will cause tilde expansion.

Tilde expansion generally occurs only at the beginning of words, but POSIX.2 has adopted an exception based on historical practice in the KornShell:

```
PATH=/posix/bin:~dgk/bin
```

is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behavior because any of the following are reasonable substitutes:

```
PATH=$(printf %s: ~rms/bin ~bfox/bin ...)
PATH=$(printf %s ~karels/bin : ~bostic/bin)
for Dir in ~maart/bin ~srb/bin ...
do
        PATH=${PATH:+$PATH:}$Dir
done
```

(In the first command, any number of directory names are concatenated and separated with colons, but it may be undesirable to end the variable with a colon because this is an obsolescent means to include dot at the end of the **PATH**. In the second, explicit colons are used for each directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.)

The exception was included to avoid breaking numerous KornShell scripts and interactive users and despite the fact that variable assignments in scripts derived from other systems will have to use quoting in some cases to allow literal tildes in strings. (This latter problem should be relatively rare because only tildes preceding known login names in unquoted strings are affected.)

Note that expressions in operands such as

```
make -k mumble LIBDIR=~chet/lib
```

do not qualify as shell variable assignments and tilde expansion is not performed (unless the command does so itself, which `make` does not).

In an earlier draft, tilde expansion occurred following any unquoted equals-sign or colon, but this was removed because of its complexity and to avoid breaking commands such as:

```
rcp hostname:~marc/.profile .
```

A suggestion was made that the special sequence ‘`$~`’’ should be allowed to force tilde expansion anywhere. Since this is not historical practice, it has been left for future implementations to evaluate. (The description in 3.2 requires that a dollar-sign be quoted to represent itself, so the `$~` combination is already unspecified.)

The results of giving tilde with an unknown login name are undefined because the KornShell `~+` and `~-` constructs make use of this condition, but in general it is an error to give an incorrect login name with tilde. The results of having **HOME** unset are unspecified because some historical shells treat this as an error.

3.6.2 Parameter Expansion

The format for parameter expansion is as follows:

`${`*expression*`}`

where *expression* consists of all characters until the matching `}`. Any `}` escaped 2
by a backslash or within a quoted string, and characters in embedded arithmetic 2
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 3.1.5), 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 3.5.2.

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 1
this subclause and in 3.2.3. (For example, `${foo-bar}xyz}` would result in the 1
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 shall exit with a nonzero exit status. Otherwise, the value of *parameter* shall be substituted. An interactive shell need not exit.

`${`*parameter*`:+`*word*`}` **Use Alternate 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 results in a test for a parameter that is unset or null; omission of the colon results in a test for a parameter that is only unset.

`${#`*parameter*`}` **String Length.** The length in characters of the value of *parameter*. 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 3.13), 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 1 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 then shall 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 then shall 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 then shall 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 then shall result in *parameter*, with the largest portion of the prefix matched by the *pattern* deleted.

3.6.2.1 Parameter Expansion Rationale. *(This subclause is not a part of P1003.2)*

When the shell is scanning its input to determine the boundaries of a name, it is not bound by its knowledge of what names are already defined. For example, if `F` is a defined shell variable, the command `"echo $Fred"` does not echo the value of `$F` followed by `red`; it selects the longest possible valid name, `Fred`, which in this case might be unset.

The rule for finding the closing `}` in `${...}` is the one used in the KornShell and is upward compatible with the Bourne shell, which does not determine the closing `}` until the word is expanded. The advantage of this is that incomplete expansions, such as

```
${foo
```

can be determined during tokenization, rather than during expansion.

The four expansions with the optional colon have been hard to understand from the historical documentation. The following table summarizes the effect of the colon:

	***parameter* set and not null**	***parameter* set but null**	***parameter* 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 *parameter*	assign *word*	
`${`*parameter* `:?`*word*`}`	substitute *parameter*	error, exit	error, exit	
`${`*parameter* `?`*word*`}`	substitute *parameter*	substitute null	error, exit	
`${`*parameter* `:+`*word*`}`	substitute *word*	substitute null	substitute null	1
`${`*parameter* `+`*word*`}`	substitute *word*	substitute *word*	substitute null	1

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.

The string length and substring capabilities were included because of the demonstrated need for them, based on their usage in other shells, such as C-shell and KornShell.

Historical versions of the KornShell have not performed tilde expansion on the word part of parameter expansion; however, it is more consistent to do so.

Examples

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

In this example, `ls` is executed only if `x` is null or unset. [The `$(ls)` command substitution notation is explained in 3.6.3.]

```
${x:-$(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.

3.6.3 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 3.12) 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, during field splitting, they may be translated into <space>s, 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

$ ‘ \

(dollar-sign, backquote, backslash). The search for the matching backquote shall 2
be satisfied by the first backquote found without a preceding backslash; during 2
this search, if a nonescaped backquote is encountered within a shell comment, a 2
here-document, an embedded command substitution of the $(*command*) form, or 2
a quoted string, undefined results occur. A single- 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 2
be used for *command* , except: 2

— A script consisting solely of redirections produces unspecified results.

— See the restriction on single subshells described below.

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 (i.e., separate them with white space).

3.6.3.1 Command Substitution Rationale. *(This subclause is not a part of P1003.2)*

The new `$( )` form of command substitution was adopted from the KornShell to solve a problem of inconsistent behavior when using backquotes. For example:

Command	Output
`echo '\$x'`	`\$x`
``echo `echo '\$x'` ``	`$x`
`echo $(echo '\$x')`	`\$x`

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the new `$( )` form can process any kind of valid embedded script, the backquoted cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

```
echo `                          echo $(
cat <<\eof                      cat <<\eof
a here-doc with `               a here-doc with )
eof                             eof
`                               )

echo `                          echo $(
echo abc # a comment with `     echo abc # a comment with )
`                               )

echo `                          echo $(
echo '`'                        echo ')'
`                               )
```

Some historical KornShell implementations did not process the first two examples correctly, but the author has agreed to make the appropriate modifications to do so. The KornShell will also be modified so that the following works: 2

```
echo $(                                    2
        case word in                       2
                [Ff]oo) echo found foo ;;  2
        esac                               2
)                                          2
```

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts. Because of its widespread historical 2
use, particularly by interactive users, however, the backquotes were retained in POSIX. 2 without being declared obsolescent.

The KornShell feature:

> If *command* is of the form <*word* , *word* is expanded to generate a pathname, and the value of the command substitution is the contents of this file with any trailing `<newline>`s deleted.

was omitted from this standard because `$(cat word)` is an appropriate substitute. However, to prevent breaking numerous scripts relying on this feature, it is 2
unspecified to have a script within `$( )` that has only redirections. 2

The requirement to separate `$(` and `(` when a single subshell is command-substituted is to avoid any ambiguities with Arithmetic Expansion. See 3.6.4.1.

3.6.4 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 2.9.2.1, with the following exceptions:

(1) Only integer arithmetic is required.

(2) The `sizeof()` operator and the prefix and postfix `++` and `--` operators are not required.

(3) Selection, Iteration, and Jump Statements are not supported.

As an extension, the shell may recognize arithmetic expressions beyond those listed. If the expression is invalid, the expansion fails and the shell shall write a message to standard error indicating the failure.

3.6.4.1 Arithmetic Expansion Rationale. *(This subclause is not a part of P1003.2)*

Numerous ballots were received objecting to the inclusion of the `(( ))` form of KornShell arithmetic in previous drafts. The developers of the standard concluded that there is a strong desire for some kind of arithmetic evaluator to replace `expr`, and that tying it in with `$` makes it fit in nicely with the standard shell language, and provides access to arithmetic evaluation in places where accessing a utility would be inconvenient or clumsy.

Following long debate by interested members of the balloting group, the syntax and semantics for arithmetic were changed. The language is essentially a pure arithmetic evaluator of constants and operators (excluding assignment) and represents a simple subset of the previous arithmetic language [which was derived from the KornShell's `(( ))` construct]. The syntax was changed from that of a command denoted by `((`*expression*`))`, to an expansion denoted by `$((`*expression*`))`. The new form is a dollar expansion (`$`), which evaluates the expression and substitutes the resulting value. Objections to the previous style of arithmetic included that it was too complicated, did not fit in well with the shell's use of variables, and the syntax conflicted with subshells. The justification for the new syntax is that the shell is traditionally a macro language, and if a new feature is to be added, it should be done by extending the capabilities presented by the current model of the shell, rather than by inventing a new one outside the model: adding a new dollar expansion was perceived to be the most intuitive and least destructive way to add such a new capability.

In Drafts 9 and 10, a form `$[`*expression*`]` was used. It was functionally equivalent to the `$(( ))` of the current text, but objections were lodged that the 1988 KornShell had already implemented `$(( ))` and there was no compelling reason to invent yet another syntax. Furthermore, the `$[]` syntax had a minor incompatibility involving the patterns in `case` statements.

The portion of the C Standard {7} arithmetic operations selected corresponds to the operations historically supported in the KornShell.

A simple example using arithmetic expansion:

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

It was concluded that the `test` command (`[`) was sufficient for the majority of relational arithmetic tests, and that tests involving complicated relational expressions within the shell are rare, yet could still be accommodated by testing the value of `$(())` itself. For example:

```
# a complicated relational expression
while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) –ne 0 ]
```

or better yet, the rare script that has many complex relational expressions could define a function like this:

```
val() {
        return $((!$1))
}
```

and complicated tests would be less intimidating:

```
while val $(( (($x + $y)/($a * $b)) < ($foo*$bar) ))
do
        # some calculations
done
```

Another suggestion was to modify `true` and `false` to take an optional argument, and `true` would exit true only if the argument is nonzero, and `false` would exit false only if the argument is nonzero. The suggestion was not favorably received by the balloting group (those contacted were negative about it, all others were silent in their latest ballots).

```
while true $(($x > 5 && $y <= 25))
```

There is a minor portability concern with the new syntax. The example `$((2+2))` could have been intended to mean a command substitution of a utility named `2+2` in a subshell. The developers of POSIX.2 consider this to be obscure and isolated to some KornShell scripts [because `$( )` command substitution existed previously only in the KornShell]. The text on Command Substitution has been changed to require that the `$(` and `(` be separate tokens if this usage is needed.

An example such as

```
echo $((echo hi);(echo there))
```

should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. However, as indicated by 3.1.1, a conforming application must separate two adjacent parentheses with white space to indicate nested subshells.

3.6.5 Field Splitting

After parameter expansion (3.6.2), command substitution (3.6.3), and arithmetic expansion (3.6.4) 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 `<space>`, `<tab>`, and `<newline>`, or if it is unset, any sequence of `<space>`, `<tab>`, or `<newline>` characters at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall delimit a field. (For example, the input

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

yields two fields, `foo` and `bar`).

(2) If the value of **IFS** is null, no field splitting shall be performed.

(3) Otherwise, the following rules shall be applied in sequence. The term "**IFS** white space" is used to mean any sequence (zero or more instances) of white-space characters that are in the **IFS** value (e.g., if **IFS** contains `<space><comma><tab>`, any sequence of `<space>` and `<tab>` characters is considered **IFS** white space).

(a) **IFS** white space shall be ignored at the beginning and end of the input.

(b) Each occurrence in the input of an **IFS** character that is not **IFS** white space, along with any adjacent **IFS** white space, shall delimit a field, as described previously.

(c) Nonzero-length **IFS** white space shall delimit a field.

3.6.5.1 Field Splitting Rationale. *(This subclause is not a part of P1003.2)*

The operation of field splitting using **IFS** as described in earlier drafts was based on the way the KornShell splits words, but is incompatible with other common versions of the shell. However, each has merit, and so a decision was made to allow both. If the **IFS** variable is unset, or is `<space><tab><newline>`, the operation is equivalent to the way the System V shell splits words. Using characters outside the `<space><tab><newline>` set yields the KornShell behavior, where each of the non-`<space><tab><newline>` characters is significant. This behavior, which affords the most flexibility, was taken from the way the original `awk` handled field splitting.

The (3) rule can be summarized as a pseudo ERE:

```
(s*ns*|s+)
```

where `s` is an **IFS** white-space character and `n` is a character in the **IFS** that is not white space. Any string matching that ERE delimits a field, except that the `s+` form does not delimit fields at the beginning or the end of a line. For example, if **IFS** is `<space><comma>`, the string

```
<space><space>red<space><space>,<space>white<space>blue
```

yields the three colors as the delimited fields.

3.6.6 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 3.13, qualified by the rules in 3.13.3.

3.6.7 Quote Removal

The quote characters

`\ ' "`

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

3.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see 3.12) or for any command. *Redirection operators* can be used with numbers representing file descriptors (see the definition in POSIX.1 {8}) as described below. See also 2.9.1. The relationship between these file descriptors and access to them in a programming language is specified in the language binding for that language to this standard.

The overall format used for redirection is:

[*n*]*redir-op word*

The number *n* is an optional decimal number designating the file descriptor number; it shall be 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 shall be 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).

In this standard, open files are represented by decimal numbers starting with zero. It is implementation defined what the largest value can be; however, all implementations shall support at least 0 through 9 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 to 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 noninteractive shell; an interactive shell may perform it, but shall do so only when the expansion would result in one word.