-- --

IEEE P1003.2 Draft 11.2 – September 1991

4.22 `expr` — Evaluate arguments as an expression

4.22.1 Synopsis

`expr` *operand* ...

4.22.2 Description

The `expr` utility shall evaluate an expression and write the result to standard output.

4.22.3 Options

None.

4.22.4 Operands

The single expression evaluated by `expr` shall be formed from the operands, as described in 4.22.7. Each of the expression operator symbols:

`(   )   |   &   =   >   >=   <   <=   !=   +   −   *   /   %   :`

and the symbols *integer* and *string* in the table shall be provided by the application as separate arguments to `expr`.

4.22.5 External Influences

4.22.5.1 Standard Input

None.

4.22.5.2 Input Files

None.

4.22.5.3 Environment Variables

The following environment variables shall affect the execution of `expr`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions and by the string comparison operators.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes within regular expressions.

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

4.22.5.4 Asynchronous Events

Default.

4.22.6 External Effects

4.22.6.1 Standard Output

The `expr` utility shall evaluate the expression and write the result to standard output. The character `'0'` shall be written to indicate a zero value and nothing shall be written to indicate a null string.

4.22.6.2 Standard Error

Used only for diagnostic messages.

4.22.6.3 Output Files

None.

4.22.7 Extended Description

The formation of the expression to be evaluated is shown in Table 4-7. The symbols *expr*, *expr1*, and *expr2* represent expressions formed from *integer* and *string* symbols and the expression operator symbols (all separate arguments) by recursive application of the constructs described in the table. The expressions in Table 4-7 are listed in order of increasing precedence, with equal-precedence operators grouped between horizontal lines. All of the operators shall be left-associative.

Table 4-7 – `expr` Expressions

Expression	Description
expr1 | *expr2*	Returns the evaluation of *expr1* if it is neither null nor zero; otherwise, returns the evaluation of *expr2*.
expr1 & *expr2*	Returns the evaluation of *expr1* if neither expression evaluates to null or zero; otherwise, returns zero.
	Returns the result of a decimal integer comparison if both arguments are integers; otherwise, returns the result of a string comparison using the locale-specific collation sequence. The result of each comparison shall be 1 if the specified relation is true, or 0 if the relation is false.
expr1 = *expr2*	Equal.
expr1 > *expr2*	Greater than.
expr1 >= *expr2*	Greater than or equal.
expr1 < *expr2*	Less than.
expr1 <= *expr2*	Less than or equal.
expr1 != *expr2*	Not equal.
expr1 + *expr2*	Addition of decimal integer-valued arguments.
expr1 – *expr2*	Subtraction of decimal integer-valued arguments.
expr1 * *expr2*	Multiplication of decimal integer-valued arguments.
expr1 / *expr2*	Integer division of decimal integer-valued arguments, producing an integer result.
expr1 % *expr2*	Remainder of integer division of decimal integer-valued arguments.
expr1 : *expr2*	Matching expression. See 4.22.7.1.
(expr)	Grouping symbols. Any expression can be placed within parentheses. Parentheses can be nested to a depth of {EXPR_NEST_MAX}.
integer	An argument consisting only of an (optional) unary minus followed by digits.
string	A string argument. See 4.22.7.2.

4.22.7.1 Matching Expression

The `':'` matching operator shall compare the string resulting from the evaluation of *expr1* with the regular expression pattern resulting from the evaluation of *expr2*. Regular expression syntax shall be that defined in 2.8.3 (Basic Regular Expressions), except that all patterns are ‘‘anchored’’ to the beginning of the string (that is, only sequences starting at the first character of a string shall be matched by the regular expression) and, therefore, it is unspecified whether ˆ is a special character in that context. Usually, the matching operator shall return a string representing the number of characters matched (`"0"` on failure). Alternatively, if the pattern contains at least one regular expression subexpression [`\(...\)`], the string corresponding to `\1` shall be returned (see 2.8.3.3).

4.22.7.2 String Operand

A string argument is an argument that cannot be identified as an *integer* argument or as one of the expression operator symbols shown in 4.22.4.

The use of string arguments `length`, `substr`, `index`, or `match` produces unspecified results.

4.22.8 Exit Status

The `expr` utility shall exit with one of the following values:

- 0 If the *expression* evaluates to neither null nor zero.
- 1 If the *expression* evaluates to null or zero.
- 2 For invalid *expression* s.
- >2 An error occurred.

4.22.9 Consequences of Errors

Default.

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

Examples, Usage

The `expr` utility has a rather difficult syntax:

— Many of the operators are also shell control operators or reserved words, so they have to be escaped on the command line.

— Each part of the expression is composed of separate arguments, so liberal usage of `<blank>`s is required. For example:

Invalid	Valid
`expr 1+2`	`expr 1 + 2`
`expr "1 + 2"`	`expr 1 + 2`
`expr 1 + (2 * 3)`	`expr 1 + \( 2 \* 3 \)`

In many cases, the arithmetic and string features provided as part of the shell command language are easier to use than their equivalents in `expr`; the utility was retained by POSIX.2 as acknowledgment of the many historical shell scripts that use it. Newly written scripts should avoid `expr` in favor of the new features within the shell.

The following command

```
a=$(expr $a + 1)
```

adds 1 to the variable **a**. A new application should use 1

```
a=$(($a+1))
```
1

The following command, for `$a` equal to either `/usr/abc/file` or just `file`:

```
expr $a : '.*/\(.*\)' \| $a
```

returns the last segment of a pathname (i.e., `file`). Applications should avoid the character `/` used alone as an argument: `expr` may interpret it as the division operator.

The following command:

```
expr "//$a" : '.*/\(.*\)'
```

is a better representation of the previous example. The addition of the `//` characters eliminates any ambiguity about the division operator and simplifies the whole expression. Also note that pathnames may contain characters contained in the **IFS** variable and should be quoted to avoid having `$a` expand into multiple arguments.

The following command

```
expr "$VAR" : '.*'
```

returns the number of characters in **VAR**.

Usage Warning: After argument processing by the shell, `expr` is not required to be able to tell the difference between an operator and an operand except by the value. If `$a` is `=`, the command:

```
expr $a = '='
```

looks like:

```
expr = = =
```

as the arguments are passed to `expr` (and they all may be taken as the `=` operator). The following works reliably:

```
expr X$a = X=
```

Also note that this standard permits implementations to extend utilities. The `expr` utility permits the integer arguments to be preceded with a unary minus. This means that an integer argument could look like an option. Therefore, the portable application must employ the `"--"` construct of Guideline 10 (see 2.10.2) to protect its operands if there is any chance the first operand might be a negative integer (or any string with a leading minus).

History of Decisions Made

In an earlier draft, Extended Regular Expressions were used in the matching expression syntax. This was changed to the Basic variety to avoid breaking historical applications.

The use of a leading circumflex in the regular expression is unspecified because many historical implementations have treated it as special, despite their system documentation. For example,

```
expr foo : ^foo                expr ^foo : ^foo
```

return 3 and 0, respectively, on those systems; their documentation would imply the reverse. Thus, the anchoring condition is left unspecified to avoid breaking historical scripts relying on this undocumented feature.

4.23 `false` — Return false value

4.23.1 Synopsis

`false`

4.23.2 Description

The `false` utility shall return with a nonzero exit code.

4.23.3 Options

None.

4.23.4 Operands

None.

4.23.5 External Influences

4.23.5.1 Standard Input

None.

4.23.5.2 Input Files

None.

4.23.5.3 Environment Variables

None.

4.23.5.4 Asynchronous Events

Default.