NAME

expr — evaluate expression

SYNOPSIS

expr (*expr* **)** [**op** *expr*]...
expr + *argument*, **match** *string regex*, **length** *string*, **index** *string characters*, **substr** *string position length* [**op** *expr*]...
expr *string* **:** *regex* [**op** *expr*]...
expr *integer* {*, **/**, **%**} *integer* [**op** *expr*]...
expr *integer* {**+**, **−**} *integer* [**op** *expr*]...
expr *expr* {**<**, **<=**, **=**, **!=**, **>=**, **>**} *expr* [**op** *expr*]...
expr *expr* **&** *expr* [**op** *expr*]...
expr *expr* **|** *expr* [**op** *expr*]...

DESCRIPTION

Writes the evaluation of the expression given as the arguments, followed by a newline, to the standard output stream. Many of the operators (**()***<>**&|**) are special in shells — make sure to escape or stringify them.

An expression qualifies as a number if it's a signed 64-bit integer ([*−9223372036854775808*, *9223372036854775807*]), decimal, with only the optional '−' allowed.

All indices are **1**-based according to characters in the current locale. Each invalid multi-byte sequence is a separate character, but regular expressions stop matching at invalid sequences.

Operators

In chunked descending precedence; all binary operators left-associative.

(*expr* **)**	*expr*
+ *argument*	Special case: immediately consumes *argument* (the next token) as a value, regardless of any special meaning.
match *string regex*	*string* **:** *regex*
length *string*	Character count in *string*.
index *string characters*	The first position in *string* of any character from *characters*, or **0** if none.
substr *string position length*	[*position*, *position* + *length*] subsection of *string*. Empty if *position* or *length* are ≤ **0** or not integers.
string **:** *regex*	The length, of the match of the basic regular expression *regex* matched to *string*, anchored to the beginning (i.e. *regex* must match the start of *string* — this is similar to prepending a "^" to *regex*), or **0** if none. If *regex* has a capture group, evaluates to the first capture group (\1), or the null string if the match failed, instead.
int * *int*	Product of *int*s.
int-l **/** *int-r*	*int-l* divided by *int-r*.
int-l **%** *int-r*	Remainder from division of *int-l* by *int-r*.
int **+** *int*	Sum of *int*s.
int-l **−** *int-r*	*int-r* subtracted from *int-l*.
expr **<** *expr*	
expr **<=** *expr*	
expr **=** *expr*	

expr **!=** *expr*	
expr **>=** *expr*	
expr **>** *expr*	If both expressions are integers, the result (**0** or **1**) of the corresponding comparison. Otherwise, the result of the corresponding comparison between the strings according to the current locale's collating sequence (dictionary order).
expr-l **&** *expr-r*	If neither expression is the null string or **0**: *expr-l*. Otherwise **0**.
expr-l **|** *expr-r*	If *expr-l* is neither the null string nor **0**: *expr-l*. Otherwise, if *expr-r* isn't the null string: *expr-r*. Otherwise **0**.
expr-l **&** *expr-r*	*expr-l* if neither expression is the null string or **0**; otherwise **0**.
expr-l **|** *expr-r*	*expr-l* if neither the null string nor **0**; otherwise *expr-r* if not the null string; otherwise **0**.

ENVIRONMENT

`EXPR_DUMP` If set, writes the final parse tree with parentheses around every expression, to the standard error stream. This is a debugging feature and will be removed.

EXIT STATUS

0 The expression evaluated to neither the null string nor **0**.
1 The expression evaluated to the null string or **0**.
2 Syntax error in expression, non-integer passed to an arithmetic operator, or division by zero.
3 Arithmetic overflow in *, **+**, or **−**.

EXAMPLES

```
$ expr 2 + 2 \* 2
6
$ expr \( 2 \) + \( 17 \* 2 \- 30 \) \* \( 5 \) + 2 - \( 8 / 2 \) \* 4
8

$ file='Makefile'; expr "$file" : '.*/\(.*\)' \| "$file"
Makefile
$ file='/usr/src/voreutils/Makefile'; expr ...
Makefile

$ file='Makefile'; expr "$file" : '\(/\)[^/]*$' \| "$file" : '\(.*\)/' \| '.'
.
$ file='/Makefile'; expr ...
/
$ file='/usr/src/voreutils/Makefile'; expr ...
/usr/src/voreutils

# However
$ file='length'; expr "$file" : '.*/\(.*\)' \| "$file"
expr: .*/\(.*\): extraneous token
$ file='length'; expr + "$file" : '.*/\(.*\)' \| + "$file"
length
```

As part of a sh(1) program:

```
#!/bin/sh
expr $# \<= 5 >/dev/null || {
    echo "$0: Too many arguments" >&2
    exit 1
}
```

SEE ALSO

Most arithmetic operations can be done using a `sh`(1) arithmetic expression (`$(( `*`expr`*` ))`), and basic string manipulation with parameter expansion operators (the `basename`(1)-like above can be written as `${file##*/}`, **`length`** `"$var"` is `${#var}`, &c.); these should be preferred for simple uses in new applications, as they're built into the shell and avoid unary operator SNAFUs.

`test`(1), `strcoll`(3), `mbrtowc`(3), `locale`(7), `regex`(7)

STANDARDS

Conforms to IEEE Std 1003.1-2024 ("POSIX.1"); **`length`**, **`substr`**, **`index`**, and **`match`** are explicitly unspecified, for compatibility with Version 7 AT&T UNIX, and are scarcely supported in non-AT&T UNIX **`expr`**s (NetBSD supports **`length`**, citing GNU system compatibility; the list ends here). Unary **`+`** is an extension, originating from the GNU system.

Some **`expr`** implementations accept flags (like FreeBSD's **`-e`**) — be wary of the first argument starting with a **`-`**, or start the argument list with a **`--`**.

HISTORY

Appears in The Programmer's Workbench (PWB/UNIX) User's Manual, allowing **`()`**, **`|&+-*/%`**, **`substr`**, **`length`**, and **`index`**, with the binary operators corresponding solely to their C equivalents on 16-bit *`int`*s.

Edition 2.3 of The CB-UNIX Programmer's Manual sees 32-bit numbers, **`|`**, **`&`**, {**`=`**, **`>`**, **`>=`**, **`<`**, **`<=`**, **`!=`**}, **`+-`**, **`*/%`**, and **`:`**, with **`substr`**, **`length`**, and **`index`** listed as **ARCHAIC FORMS**. **`|`** is described simpler, as *`expr-l`* if not nullary and *`expr-r`* otherwise (with no **0**-folding), but the global behaviour is described as

> Note that **0** is returned to indicate a zero value, rather than the null string.

The present-day behaviour matches and falls out of this. The comparison operators for non-integers are byte-wise, owing to no system localisation. **`:`** rejects patterns with more than one capture group, but is otherwise as present-day. Integer arguments to **`substr`** now default to **0** instead of being required to be integers.

IEEE Std 1003.1-2008 ("POSIX.1") notes that on some systems **`:`** is documented as literally injecting a **`^`**, supposedly making another one in the pattern plain text, despite not doing so and selecting the match some other way — this is the case here. Of interest is also that the **ARCHAIC FORMS** are such because they "have been made obsolete by the : operator" — the suggested replacements are:

`substr` *`abcd 2 2`*	*`abcd`* **`:`** *`'..\(..\)'`* — this is mostly reasonable, but more accurate as *`'..\(..\?\)'`*, and more generic as *`'.\{2\}\(.\{1,2\}\)'`*.
`length` *`expr`*	*`expr`* **`:`** *`'.*'`*
`index` *`abcd d`*	*`abcd`* **`:`** *`d`*. Not even close! This is approximately seven centimeters down from explaining how **`:`** is anchored and what that entails. Recreating **`index`** is very likely impossible with **`:`**, even for a simple single-letter case.

`match` is also available, but wholly undocumented.

AT&T System III UNIX inherits the CB-UNIX manual page but strips it of the unary operators.

AT&T System V UNIX removes **`substr`**, **`length`**, and **`index`**.

Version 7 AT&T UNIX, on the other hand, sees an **`expr`** compatible with CB-UNIX's, but with an unrelated manual page, not mentioning the unary operators at all.

4.4BSD errors on **`/%`** dividing by zero instead of performing the division (which resolves to zero on the PDP-11 but a `SIGFPE` on the VAX).