NAME

hoc — (high-order calculator) [interactive floating-point language]

SYNOPSIS

hoc [**−author**] [**−copyright**] [**−?**] [**−help**] [**−no-banner**] [**−no-help-file**] [**−no-load**]
[**−no-logfile**] [**−no-readline**] [**−no-save**] [**−no-site-file**] [**−no-translation-file**] [**−no-user-file**]
[**−quick**] [**−silent**] [**−version**] [*file* . . .]

OPTIONS

hoc options can be prefixed with either one or two hyphens, and can be abbreviated to any unique prefix. Thus, **−a**, **−author**, and **−−auth** are equivalent.

To avoid confusion with options, if a filename begins with a hyphen, it must be disguised by a leading absolute or relative directory path, e.g., */tmp/-foo.hoc* or *./-foo.hoc*.

−author Display an author credit, and software distribution information, on the standard error unit, *stderr*, and then terminate with a success return code. Sometimes an executable program is separated from its documentation and source code; this option provides a way to recover from that.

−copyright Display copyright information on the standard error unit, *stderr*, and then terminate with a success return code.

−help or **−?** Display a help message on *stderr*, giving a brief usage description, and then terminate with a success return code.

−no-banner Suppress any welcome banners normally printed by dynamically-loaded library code.

This option can also be set via the **hoc** system variable **__BANNER__**, but it must be set in an initialization file before code in that file to print the welcome banner is reached.

−no-help-file Suppress loading of system-wide **hoc** help files at startup.

−no-load Disable the **load( )** function. It will continue to be recognized, but when invoked, will simply print a warning that it has been disabled.

This option is a security feature: it takes effect only after all initialization files have been processed.

−no-logfile Disable the **logfile( )**, **logon( )**, and **logoff( )** functions. They will continue to be recognized, but when invoked, will simply print a warning that they have been disabled.

This option is a security feature: it takes effect only after all initialization files have been processed.

−no-readline Suppress use of the GNU *readline* library: command completion, editing and recall are then not available.

On some systems, it may be necessary to use this option when **hoc** is used in international mode (see the **INTERNATIONALIZATION** section below) in order to get accented letters displayed properly.

−no-save Disable the **save( )** function. It will continue to be recognized, but when invoked, will simply print a warning that it has been disabled.

This option is a security feature: it takes effect only after all initialization files have been processed.

−no-site-file Suppress loading of the system-wide non-help startup files.

−no-translation-file Suppress loading of the system-wide message translation files.

−no-user-file Suppress loading of the user-specific startup file.

−quick Suppress loading of all startup files: this option is equivalent to **−no-help-file −no-site-file −no-translation-file −no-user-file**.

−silent Suppress printing of prompts for interactive input. **hoc** never prompts when it is reading noninteractive files.

The **hoc** system variable **__VERBOSE__** can also be set to zero at run time to turn off prompts; setting it to nonzero turns them back on.

The **hoc** system variable **__PROMPT__** contains the prompt string: it can be redefined at any time.

−version Display the program version number and release date on *stderr*, and then terminate with a success return code.

DESCRIPTION

hoc interprets a simple language for floating-point arithmetic, at about the level of Basic, with C-like syntax and functions. However, unlike Basic, **hoc** has particularly rich support for floating-point arithmetic, and its facilities are certainly better than that standardly provided by most programming languages, such as C, C++, and Fortran.

To get a flavor of what typical **hoc** code looks like, visit the `*.hoc` and `*.rc` files in the **hoc** installation directory tree: see the **INITIALIZATION FILES** section below for their location.

The named *file*s are read and interpreted in order. If no *file* is given or if *file* is −, **hoc** interprets the standard input.

hoc input consists of *expressions* and *statements*. Expressions are evaluated and their results printed. Statements, typically assignments and function or procedure definitions, produce no output unless they explicitly call **print**.

Word completion

When **hoc** has been built with the GNU *readline* library, word completion can be used to save typing effort. It is normally requested by an ESCape character following a prefix of a word in **hoc**'s symbol table: **hoc** will respond with an audible beep, and a list of words that match that prefix, or if only one word matches, it will silently complete the word:

```
hoc> c<ESCape>
cbrt ceil copysign cos cosd cosh
hoc> co<ESCape>
copysign cos cosd cosh
hoc> cop<ESCape>
hoc> copysign
```

The character used to request completion can be changed: see the **INITIALIZATION FILES** section below.

Command history and editing

When **hoc** has been built with the GNU *readline* library, convenient command history and editing support is available, much like it is in the UNIX **bash**(1), **ksh**(1) and **tcsh**(1) shells, and in a few GNU programs, like the **bc**(1) and **genius**(1) calculators. The default history and editing mode is **emacs**(1)-style; you can also get **vi**(1)-style by suitable customizations: see the **INITIALIZATION FILES** section below.

In the default mode, *C-p* (hold the Control key down while typing p) moves up in the history list, *C-n* moves down, *C-b* moves backward in the current line, *C-f* moves forward, *C-d* deletes forward, *DELete* deletes backward, *C-a* moves to the beginning of the line, *C-e* moves to the end of the line, *C-l* repaints the screen, reprinting the current line at the top, and *RETurn* resubmits the line for execution.

For more details, consult the *GNU Readline Library* manual, available online in the *info* system. In **emacs**(1); type *C-h i mreadline* to get there.

Numbers

All numbers in **hoc** are stored as double-precision floating-point values.

On systems with IEEE 754 arithmetic, such numbers are capable of representing integers of up to 53 bits

exactly, excluding the sign bit. This is an integer range of −(2^53) ... 2^53, or −9,007,199,254,740,992 ... 9,007,199,254,740,992.

Numbers may be signed, and may optionally contain a decimal point, and a power-of-ten exponent, which consists of the letter **e** (or **E**) followed by an optionally-signed integer. No other exponent letters are recognized.

A hexadecimal floating-point number format, introduced in the latest ISO C Standard, *ISO/IEC 9899:1999 (E) Programming languages — C*, usually known by its short name, *C99*, is also supported, and implemented by portable private code in **hoc**. This format consists of an optional sign, then *0x* or *0X*, followed by one or more hexadecimal digits (*0–9 A–F a–f*) containing at most one hexadecimal point, optionally followed by a binary (power-of-two) exponent consisting of *p* or *P* followed by an optionally-signed decimal integer. Thus, *−0x1.00000p8*, *−0x100*, *−0x100000p-12*, *−0x10p+4*, *−0x1p+8*, *−0x1p00008*, and *−0x1p8* all represent the decimal number *−256*.

The hexadecimal format, while awkward for humans, has the advantage of guaranteeing exact input/output conversions on all platforms, and **hoc** consequently uses this format in files created by the **save( )** command.

Strings

String constants are delimited by quotation marks (**" . . . "**), and may not span multiple lines, unless the embedded line breaks are each prefixed with a backslash, which is removed, leaving the newline in the string.

All characters in 1 ... 255 are representable in strings; as in C and C++, character 0 (ASCII NUL) is reserved as a string terminator.

In string constants, nonprintable characters may be represented by the usual escape sequences defined in Standard C and Standard C++, plus one extension (**\E**):

\\	backslash: ASCII decimal 92.
\"	quotation mark: ASCII decimal 34.
\a	alert or bell (ASCII BEL: decimal 7).
\b	backspace (ASCII BS: decimal 8).
\E	escape (ASCII ESC: decimal 27).
\f	formfeed (ASCII FF or NP: decimal 12).
\n	newline (ASCII LF or NL: decimal 10).
\r	carriage return (ASCII CR: decimal 13).
\t	horizontal tab (ASCII HT: decimal 9).
\v	vertical tab (ASCII VT: decimal 11).
\o \oo \ooo	octal character number (**o** = *0–7*) in one to three digits.
\xh...	hexadecimal character (**h** = *0–9A–F* or *0–9a–f*) in one or more digits.

Backslash followed by any other character than those listed is simply discarded: **W** reduces to **W**.

Variables

Variable names consist of an initial letter or underscore, followed by any number of letters, underscores, or digits. Lettercase is *significant*. Letters are considered to be *A–Z*, *a–z*, and any characters in the range 160 ... 255 of an 8-bit character set. Use of characters in the latter range is normally not recommended, because they are often difficult, or impossible, to generate on some computer keyboards. Nevertheless, it does permit non-English words to be spelled correctly; see the **INTERNATIONALIZATION** section below.

Underscore (_) by itself is a reserved variable containing the value of the last *numeric* expression evaluated. Double underscore (__) is a reserved variable containing the value of the last *string* expression evaluated. They cannot be assigned to by user code.

Predefined numeric constants and variables

Certain immutable named constants are already initialized:

CATALAN	Catalan's constant: sum((−1)^i/(2*i+1)^2, i = 0..infinity) = approximately 0.915965594177219015054603514932. . .
DEG	180/PI, degrees per radian
E	base of natural logarithms
GAMMA	Euler's constant: limit(sum(1/i,i=1. . .n) − ln(n), n → infinity) = approximately 0.577215664901532860606512090082. . .
INF or **Inf** or **Infinity**	IEEE-754 floating-point infinity
MAXNORMAL	Largest finite normalized floating-point number.
MINNORMAL	Smallest (in absolute value) nonzero normalized floating-point number.
MINSUBNORMAL	Smallest (in absolute value) subnormal floating-point number. If your computer system does not support subnormal numbers, this is identical to **MINNORMAL**.
NAN or **NaN**	IEEE-754 floating-point not-a-number
PHI	golden ratio: (1 + sqrt(5))/2 = approximately 1.618033988749894848204586834 36. . .
PI	ratio of the circumference of a circle to its diameter, approximately 3.14159265358979323846264338327. . .
PREC	maximum number of significant digits in output, initially 17 on most systems (the precise value is computed dynamically, from Matula's 1968 result: *ceil(N/log_b(10) + 1)*, for a host floating-point system with *N* base-*b* digits). **PREC = 0** gives shortest 'exact' values.
QNAN or **QNaN**	IEEE-754 floating-point quiet not-a-number
SNAN or **SNaN**	IEEE-754 floating-point signaling not-a-number

More information on the floating-point constants is available in the **FLOATING-POINT ARITHMETIC** section below.

Predefined system constants and variables

hoc also provides a number of system constants and variables, adopting the C/C++ convention that names beginning with two underscores are reserved for the implementation:

__BANNER__	[reassignable number] Nonzero (true) if printing of welcome banners is permitted. It can be changed by the **−no-banner** option.
__DATE__	[constant string] Date of the start of job execution, in the form `"Dec 16 2001"`. The day number has a leading space if only one digit is needed, so that the string always has constant width.
__FILE__	[constant string] Name of the current input file. This is `"/dev/stdin"` when **hoc** is reading from the standard input.
__FILE__[*n***]**	[constant string] Name of the *n*-th input file in the current job. This provides a history of exactly what files have been read. Because **hoc** does not yet support arrays, the only way to display these is with the **who( )** function.

__IEEE_754__	[constant number] Nonzero (true) if the host system supports IEEE 754 arithmetic.
__LINENO__	[constant number] Number of the current input line in the file named by **__FILE__**.
__PACKAGE_BUGREPORT__	[constant string] Where to report bugs.
__PACKAGE_DATE__	[constant string] Date of last modification of the software.
__PACKAGE_NAME__	[constant string] Program name.
__PACKAGE_STRING__	[constant string] Program name and version number.
__PACKAGE_VERSION__	[constant string] Program version number.

__PROMPT__	[reassignable string] Current prompt string. Prompting is controlled by the setting of **__VERBOSE__** (see below).
	For example,
	`__PROMPT__ = "\n\E[7mInput:\E[0m "`
	will produce a blank line followed by a prompt in inverse video in terminal emulators, such as **xterm**(1) and DEC VT100, that follow the ANSI X3.64-1979 or ISO 6429-1983 terminal standards.
	If **__PROMPT__** contains the two-character format string %d, that string will be replaced by the prompt count: for example, this silly setting
	`__PROMPT__="\E[4;5;34;43m[%d]\E[0m: "`
	will display the count digits in blue, and underlined, on a yellow background, in an **xterm**(1) window that supports text color attributes. [Run `dircolors -p` for more information on color settings.]

__READLINE__	[constant number] Nonzero (true) if the GNU *readline* library is in use.
__TIME__	[constant string] Local time-of-day (24-hour clock) of the start of job execution, in the usual hours, minutes, seconds form `"14:57:23"`.
__VERBOSE__	[reassignable number] Nonzero (true) if **hoc** should prompt for input from interactive files. The actual prompt string is controlled by the **__PROMPT__** variable. [NB: A bug in the GNU *readline* library (version 4.2a) makes this variable ineffective; it works correctly with the **−noreadline** option. The bug has been reported to the *readline* maintainers.]

Numeric expressions

Numeric expressions are formed with these C-like operators, listed by decreasing precedence.

^	Exponentiation.
! - ++ --	Logical negation, arithmetic negation, increment-by-one, decrement-by-one. As in C and C++, the latter two may be applied *before* a variable (acting first before taking the value), or *after* (taking the current value first, then acting).
*** / %**	Multiply, divide, modulus.

`+ -`	Add, subtract.
`> >= < <= == !=`	Greater than, greater than or equal to, less than, less than or equal to, equal to, not equal to.
&&	Logical and. Both operands are *always* evaluated, unlike in C and C++, where the second is evaluated only if the first is nonzero (true).
`||`	Logical or. Both operands are *always* evaluated, unlike in C and C++, where the second is evaluated only if the first is zero (false).
`= += -= *= /= %= :=`	Assignment, assign the left-hand side the (sum, difference, product, dividend, or modulus) of its current value and the right-hand side, permanent assignment. The operator `:=` is a *one-time-only* assignment operator, used for defining permanent constants that cannot be redefined in the same **hoc** session. As in C and C++, assignment is a right-associative expression whose value is the left-hand side. This means that `x = y = z = 3` is interpreted as `x = (y = (z = 3))`. That is, `3` is assigned to `z`, then that result is assigned to `y`, and finally, that result is assigned to `x`, so all three variables are assigned the value `3`. Similarly, `sqrt(x = 4)` assigns the value `4` to `x` before computing and returning its square root.

Expression lists in **print**-like statements, and in argument lists, are evaluated in strict *left-to-right* order. Thus, the output of expressions with side effects, such as

```
n = 3
print ++n, n++
```

is predictable: that example prints

```
4 4
```

String expressions

String expressions support only the relational operators (`> >= < <= == !=`) and the simple assignment operators (`= :=`), plus concatenation, which is indicated by two successive string expressions, without any specific operator, following the practice in C, C++, and **awk**(1). These two assignments are equivalent:

```
s = "hello"   ", "   "wor"   "ld"
s = "hello, world"
```

Numbers in string expressions are converted to strings according to the current precision variable, **PREC**.

```
k = 123
PREC = 4
s = "abc" k "def" PI
println s
abc123def3.142
```

Several string functions listed below augment string expressions.

Built-in functions and procedures

Longer documentation of the built-in functions and procedures is relegated to the later section, **DESCRIPTIONS OF BUILT-IN FUNCTIONS AND PROCEDURES**.

These numeric built-in functions take zero arguments: **rand**, **second**, and **systime**.

These numeric built-in functions take one numeric argument: **abs**, **acos**, **acosh**, **asin**, **asinh**, **atan**, **atanh**, **cbrt**, **ceil**, **cos**, **cosd**, **cosh**, **erf**, **erfc**, **exp**, **expm1**, **exponent**, **factorial**, **floor**, **gamma**, **ilogb**, **int**, **isfinite**, **isinf**, **isnan**, **isnormal**, **isqnan**, **issnan**, **issubnormal**, **J0**, **J1**, **lgamma**, **ln**, **log**, **log10**, **log1p**, **log2**, **macheps**, **nint**, **number**, **randl**, **rint**, **rsqrt**, **setrand**, **significand**, **sin**, **sind**, **sinh**, **sqrt**, **tan**, **tand**, **tanh**, **trunc**, **Y0**, and **Y1**.

These numeric built-in functions take two numeric arguments: **copysign**, **errbits**, **fmod**, **gcd**, **hypot**, **Jn**, **lcm**, **ldexp**, **logb**, **max**, **min**, **randint**, **nearest**, **nextafter**, **remainder**, **scalb**, and **Yn**.

These string built-in functions take zero arguments: **logoff**, **logon**, and **now**.

These string built-in functions take one argument: **eval**, **getenv**, **length**, **hexfp**, **hexint**, **load**, **logfile**, **printenv**, **string**, **tolower**, **toupper**, and **who**.

These string built-in functions take two arguments: **index**, **putenv**, **save**, and **strftime**.

This string built-in function takes three arguments: **substr**.

These startup file procedures take no arguments: **author**, **help**, **help_xxx**, and **news**.

The help system (described later) documents each of these functions, and any additional ones provided by startup files. Most have the same names as they do in C, C++, and Fortran, so many will already be familiar to users who have learned any of those programming languages.

Built-in functions and procedures are *immutable*: they cannot be redefined by the user in **hoc** code. User-defined variables, functions, and procedures can be redefined at any time to objects of the same type. Variables can be redefined to be functions or procedures. However, the reverse does not hold: once a name has been used as a function or procedure, it can only be redefined to be a new function or procedure.

The procedure **abort(message)** prints **message**, immediately terminates evaluation, and returns to the top-level interpreter, discarding and clearing the function/procedure call stack. It is equivalent to a similar internal function that **hoc** uses to recover from catastrophic errors. Use it sparingly!

The function **read(x)** reads a value into the variable **x**. The value must be either a number, or a quoted string, or an existing variable or named constant. The return value is 1 on success, or 0 on end-of-file; the function aborts for any other error condition.

The statement **print** prints a list of expressions that may include string constants such as `"hello\n"`. It does *not* print a final newline: the last expression must end with one if a newline is required.

The statement **println** works like **print**, but always supplies a following newline.

There is an incompletely implemented **printf** statement: it will not be further documented until it is fully working.

The function **who(prefix)** produces a lengthy report of all of the named constants and variables with their current values, plus the names of all built-in functions and procedures, and all user-defined functions and procedures. Only those names whose initial letters match the argument string, **prefix**, are included. To print all symbols, use an empty prefix: **who("")**. The return value is always an empty string.

Symbols with three or more leading underscores are for internal use by **hoc**, and are thus considered *hidden*. They can only be shown by a suitable **prefix** argument to **who( )**. Hidden symbols are used for locale translations of embedded strings. See the **INTERNATIONALIZATION** section below for further details.

Statements

Control flow statements are **if**–**else**, **while**, and **for**, with braces for grouping. Newline or semicolon ends a statement. Backslash-newline is equivalent to a space.

Functions and procedures are introduced by the words **func** and **proc**; **return** is used to return with a value from a function. Within a function or procedure, arguments are referred to as **$1**, **$2**, etc.; all other variables are global.

FLOATING-POINT ARITHMETIC

All arithmetic in **hoc** is done in double-precision floating point (C/C++ type **double**).

On most modern systems, this arithmetic conforms closely (or loosely) to the 1985 *IEEE 754 Standard for Binary Floating-Point Arithmetic*. This arithmetic system has numerous advantages over older designs, and has helped enormously to improve the environment for, and portability and reliability of, numerical software.

How floating-point numbers are represented

In IEEE 754 arithmetic, double-precision numbers are represented as 64-bit values, consisting of a sign bit, an 11-bit biased exponent, and a 53-bit significand. That is a total of 65 bits: the first significand bit is called a *hidden* bit, and is not actually stored. The binary point lies between the hidden bit and the stored

fraction, so that for normal numbers, the significand is at least one, but less than two.

Biased, rather than explicitly signed, exponents are conventional in floating-point architectures. For IEEE 754 64-bit arithmetic, the exponent bias is 1023; that is, the true exponent is 1023 less than the stored biased value.

The smallest biased exponent (0), and the largest biased exponent (2^11 - 1 = 2047), are given special interpretation, described below for subnormals, and Infinity and NaN, respectively.

Large normal numbers

With the IEEE 754 format, the number range is approximately −1.80e+308 .. +1.80e+308, with a precision of about 15 decimal figures. The exact value of the largest floating-point number is **(1 − 2^(−53)) * 2^1024**.

Small normal numbers

The smallest *normalized* number that can be represented is about 2.23e-308, or more precisely, **2^(−1022)**, and its reciprocal is also representable, being almost exactly a quarter of the largest representable number.

Smaller subnormal numbers

The IEEE 754 Standard defines a numerically useful feature called *gradual underflow* that, when the biased exponent reaches its smallest value (0), relaxes the normalization requirement and drops the hidden bit, permitting small numbers to decrease further down to about 4.94e-324, or more precisely, **2^(−1074)**, but with loss of precision. Such numbers are called *subnormal* (formerly, *denormalized*). Not all systems support such numbers: the **hoc** function **issubnormal(x)** can be used to test whether **x** is subnormal. The reciprocal of the largest floating-point number is nonzero only if subnormal numbers are supported. Thus, you could define this **hoc** function to find out whether your system has subnormals; it returns 1 (true) if that is the case:

```
func hassubnormals( ) \
    return (issubnormal(1/(((1 - 2^(-53)) * 2^1023) * 2)))
```

With a predefined constant, this can also be written as

```
func hassubnormals( ) return (issubnormal(1/MAXNORMAL))
```

Underflow

Numbers below the smallest normalized, or when supported, the smallest subnormal, values quietly *underflow* to zero.

Machine epsilon

Another significant quantity in *any* floating-point system is known as the *machine epsilon*. This is the smallest positive number that can be added to one, and produce a sum still different from one. **hoc** provides a generalization of this, with **x** replacing **one** in the last sentence: **macheps(x)**.

In IEEE 754 arithmetic, **macheps(1)** is about 2.22e-16, or more precisely, **2^(−52)**. The negative of its base-10 logarithm is the number of decimal digits that can be represented. An error of **macheps(x)** is called an *ULP* (*U*nit in the *L*ast *P*lace). If **y** is an approximation to **x**, then with the definition

```
func errbits( ) \
{
    if ($1 == $2) \
        return (0) \
    else \
        return (ceil(log2(abs(($1 - $2)/max($1,$2))/macheps($1))))
}
```

errbits(x,y) is the number of bits that are in error in **y**: that is, the base-2 logarithm of the relative error in ULPs, rounded up to the nearest integer. Incidentally, this function behaves as expected if either of its arguments are NaN (described below), or Infinity of opposite signs, even though there are no tests for those values: the result is a NaN.

One might reasonably argue for **errbits(x,y)** that the case of two Infinity arguments of like sign should also return a NaN. The current implementation does not include such a test, but doing so would require just one additional statement: `if (isinf($1) && isinf($2)) return (NAN)`.

macheps(0) is the smallest representable floating-point number, either normalized, or subnormal if

supported. Thus, the test function above can be written more simply and portably (since it also works for non-IEEE 754 systems) as

```
func hassubnormals( ) return issubnormal(macheps(0))
```

but it will run somewhat more slowly, since the current portable implementation of **macheps(x)** involves a loop. Another simple implementation of this function uses predefined constants:

```
func hassubnormals( ) return (MINNORMAL > MINSUBNORMAL)
```

Special values: Infinity and NaN

IEEE 754 also defines two special values: Infinity, and NaN (not-a-number). The latter are expected to be available in two flavors: quiet and signaling, but some architectures provide only one kind. The distinction between the two NaNs is rarely significant: the Standard's intent was that quiet NaNs should be generated in numerical operations, while signaling NaNs could be used to initialize numeric variables, so that their use before assignment of a normal value could then be trapped.

Both Infinity and NaN are signed, but the sign of a NaN is usually irrelevant, and may not reflect how it was computed: some architectures only generate negative NaNs, others generate only positive ones, and a few may preserve the expected sign in the NaN produced.

Signed zero

IEEE 754 has both positive and negative zero, but they compare equal. A positive zero is represented by all zero bits. A negative zero has a leading one-bit, followed by 63 zero bits.

Negative zero is generated from

0 / −Infinity
sqrt(−0)

In principle, you should be able to get a negative zero in any programming language by simply writing **−0**, but many compilers will convert this to positive zero. You then have to introduce a variable, assign it a zero, and negate the variable, possibly hiding the negation in an external function that simply returns its value, to foil optimizers. In **hoc**, however, **−0** works correctly.

Signs of numbers

In **hoc**, you can extract the sign of any value, **x**, including negative zero, Infinity, and NaN, like this:

copysign(1,x)

The result will be either +1 or −1.

Nonstop computing

Infinity and NaN are intended to provide *nonstop computing* behavior. In contrast, older architectures tended to abruptly terminate a job that computed a number too large to be stored (an *overflow*), or divided by zero. IEEE 754 arithmetic produces Infinity or NaN for these two cases, according to well-defined, and obvious, rules discussed below.

On these older systems, **hoc** tries to prevent generation of exceptional values that might otherwise terminate the job: it aborts such computations with an error message, and returns you to top level, ready for more input. On IEEE 754 systems, computation in **hoc** simply proceeds as the Standard intended.

The IEEE 754 nonstop property is exceedingly important in modern heavily-pipelined, or parallel, or super-scalar, or vector, architectures, all of which have multiple operations underway at once. An interrupt to handle a floating-point exception in software is extremely costly in performance.

Properties of Infinity and NaN

Both Infinity and NaN propagate in computations, so that if they occur in intermediate results, they will usually be visible in the final results too, and alert the user to a potential problem.

Infinity behaves somewhat like a mathematical infinity:

finite / Infinity → 0
Infinity * Infinity → Infinity
Infinity^(finite or Infinity) → Infinity

NaN is produced whenever one or more operands of an arithmetic expression is a NaN, or from most numerical functions with NaN arguments, or from expressions where a limiting value cannot be determined:

Infinity − Infinity → NaN
Infinity / Infinity → NaN
0 / 0 → NaN

NaN has a unique property not shared by any other floating-point values, including Infinity: it is not equal to anything, even itself! This should be usable as a completely portable test for a NaN, even on older systems that do not have IEEE 754 arithmetic:

(x != x) is true if, and only if, **x** is a NaN.

Regrettably, compiler writers on several systems have failed to grasp this important point, and they incorrectly optimize this test to false. Thus, portable code needs to use a test function, and **hoc** provides three of them: **isnan(x)**, **isqnan(x)**, and **issnan(x)**, which return true if **x** is a NaN (of any flavor, or quiet, or signaling, respectively).

What NaNs mean for programmers

The presence of NaNs in the arithmetic system has an extremely important implication for numerical software: comparisons now have *three* outcomes, not two. The expression **(x < y)** will be true or false if neither **x** nor **y** is a NaN, but it is called *unordered* if either, or both, is a NaN. In particular, this means that it is almost always *wrong* to use a computer programming language two-branch **if − else** statement with a numerical test. Instead, there need to be additional initial tests to check for NaNs. Thus, instead of the **hoc** statement

```
if (x > y) \
    print "x is greater than y\n" \
else \
    print "x is less than or equal to y\n"
```

you should instead write

```
if (isnan(x)) \
    print "x is a NaN\n" \
else if (isnan(y)) \
    print "y is a NaN\n" \
else if (x > y) \
    print "x is greater than y\n" \
else \
    print "x is less than or equal to y\n"
```

Since **if − else** statements are very common in software, but most programmers, and computer textbook authors, are not sufficiently familiar with IEEE 754 arithmetic, you should expect that most existing software, and textbook examples, will fail to behave consistently, or correctly, when dealing with NaN, and possibly also Infinity.

There have been some major disasters, such as the failure of the Ariane satellite launch in West Africa, the failure of Patriot missiles in the Gulf War, and a U.S. nuclear aircraft carrier sitting dead in the water for six hours, all attributed to computer programmers who lacked sufficiently understanding of computer arithmetic. Arithmetic really does matter!

Numerical software often contains convergence tests of the form

while (tolerance is not reached)
  reduce the tolerance

If a NaN ever appears in the **while** expression, the test will never be satisfied, and the program will be in an infinite loop. Even famous libraries like EISPACK and LINPACK have routines that will never return because of loops caused by NaNs. [In fairness, both of those libraries were developed before IEEE 754 arithmetic existed, but CDC and Cray machines of that era had special values similar to Infinity and NaN, so even then, there were systems where the code could endlessly loop.]

Vendor-provided floating-point systems and run-time libraries are not always entirely reliable in their handling of signed zero, Infinity, and NaN, and portable programs like **hoc** can help to ferret out implementation differences, and errors that should be reported to the vendors. As noted earlier, signed zero is often botched by compiler writers, and two functions commonly available in most programming languages, **max(x,y)** and **min(x,y)**, in particular are badly done. Their simple implementations use a two-branch

conditional like this one for **max(x,y)**: `if (x > y) return x else return y`. If either argument is a NaN, then the test will fail, and the second argument will be returned, leading to inconsistent nonsense like **max(1,NaN)** → **NaN** but **max(NaN,1)** → **1**. The C and C++ languages lack such functions (users are expected to write them as macros), but Fortran and many other languages have them. In the fall of 2001, tests of 61 Fortran compilers on 15 different UNIX platforms showed that *all* fail to behave consistently for **max(x,y)** and **min(x,y)**.

Unsupported IEEE 754 features

Finally, there are two additional features of IEEE 754 arithmetic that are not yet supported by this version of **hoc**, but will be in future releases:

(1) access to floating-point status flags, so that you can tell after the fact whether a computation encountered any exceptional conditions, and

(2) access to rounding control, which determines whether rounding is to minus Infinity, zero, nearest, or plus Infinity. The default is always round-to-nearest.

Once rounding control is available, **hoc** could, in principle, be extended to support interval arithmetic, in which each numeric operation produces upper and lower bounds for the result. Of course, a proper implementation would also require such support in all of the mathematical functions in the C/C++ run-time library, and such support is lacking almost everywhere.

HELP SYSTEM

One of the files that **hoc** normally loads on startup contains an extensive help system. Each named constant, variable, function and procedure has an associated function, **help_NAME( )**, where **NAME** is the object name. Help is also available on each of the **hoc** language statements, and on related topics. For an introduction, run **help( )**, and for a detailed list of what help functions are available, invoke **help_help( )**. To display the entire help system, invoke **help_all( )**.

Users are encouraged to follow these help convention with their own **hoc** code.

The entire help corpus is intentionally *external* to **hoc** itself, to facilitate modification, partial replacement, and internationalization, as discussed in the next section.

INTERNATIONALIZATION

The **hoc** help system can be readily extended to support documentation in languages other than English, and early releases contain limited prototype text in several languages.

Changing the language alters only documentation and program messages: the basic **hoc** language remains unchanged, and English-centric, just as do virtually all computer programming languages.

Selecting a language

An alternate language is selected at run-time by defining any one of three environment variables: **LC_ALL**, **LC_MESSAGES**, or **LANG**, just as described for other programming languages in **locale**(1). These variables take values of a locale code, the values of which you can list by

```
locale -a | sort -f
```

You could thus launch a German version of **hoc** like this:

```
env LANG=de hoc
```

Environment variables, rather than command-line options, control the locale selection, because it is likely that most individuals will want to choose a fixed locale, and that can be done once and for all in user login files, and also because several UNIX library functions access the locale environment variables to guide their behavior. UNIX users could also create convenient shell aliases, e.g., in **csh**(1)/ **tcsh**(1) syntax,

```
alias hoc-da 'env LANG=da hoc \!*'
alias hoc-de 'env LANG=de hoc \!*'
alias hoc-fr 'env LANG=fr hoc \!*'
...
```

What if you have no locale support?

Virtually all UNIX vendors today provide locale support, but they usually require installation of one or more additional software packages that your system manager may have omitted, but is probably willing to install on request.

Locale support is usually present in one of these directories; besides using the **locale**(1) command as shown in the previous subsection, you can run **ls**(1) on the appropriate one of them to see what locales are installed on your system:

/usr/share/locale	Apple Darwin (MacOS X), FreeBSD, GNU/Linux (all architectures)
/usr/lib/nls/loc	Compaq/DEC Alpha, IBM AIX
/usr/share/i18n/locales	GNU/Linux (all architectures)
/usr/lib/nls/loc/locales	Hewlett-Packard HP-UX
/usr/lib/locale	SGI IRIX, Sun Solaris

What the locale affects

Normally, changing the locale affects more than just text: dates, monetary formats, numbers, and sort order all change. However, for now, in the interests of simplicity, and cross-platform and cross-locale consistency, **hoc** sets the locale categories for **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** to their traditional (English/American) values. Changes will be needed in future versions of **hoc** to support other values of these categories; some of that support is already available, as shown in the next subsection.

Changing the locale inside hoc programs

Locale categories can be set in the environment from *inside* **hoc** programs to control calendar date and time formatting by the **strftime( )** function:

```
# Show time in the default locale:
hoc> strftime("%c",systime())
Fri Dec 21 15:18:14 2001

# Switch to Portuguese: ISO8859-1 (Latin-1) encoding:
hoc> old_lc_time = putenv("LC_TIME", "pt")
hoc> strftime("%c",systime())
sex 21 dez 2001 03:17:29 PM MST

# Restore the original locale:
hoc> ignore = putenv("LC_TIME", old_lc_time)
```

The current locale setting can be saved and restored as shown. Less desirably, the value `"C"` resets it to the C/C++ default of English.

The locale code is interpreted as the name of a subdirectory in which to find a localized version of any system file that **hoc** loads at startup time. For example, in a Danish locale, it will load the English file, `help.hoc`, and then the Danish file, `da/help.hoc`, from the **hoc** system installation directory, provided that the localized file exists. Otherwise, **hoc** is silent about its absence.

Changing the language of internal messages

The **hoc** program contains a number of messages that are hard-coded in English. Any, or all, of these can be replaced at run time by assignments to special variables named with the reserved seven-character prefix **___msg_** (yes, there are *three* leading underscores) used to identify translation variables.

These variables are normally only set in the *translations.hoc* files in the **hoc** system directory tree, but they can also be set by user programs as well, unless they have been defined as permanent constants.

See the comments in those files for further documentation. Except for translation work, it should never be necessary for ordinary users to reference or modify these variables.

Character set constraints

The significant constraint is that characters must be representable in 8-bit character sets, such as the dozen or so ISO8859-*n* sets that supply characters needed for European languages, or the Unicode (also known as ISO10646-1) UTF-8 variable-byte-count encoding of potentially two million or so symbols used in the world's writing systems. In addition, the **hoc** user must be running the program in an environment capable of such display.

Changing screen display fonts

In a UNIX system, you might first scan the voluminous output of **xlsfonts**(1) to find out what fonts are available for your window system, and then launch a terminal window like this:

```
xterm -fn \
    -adobe-courier-medium-r-normal--14-100-100-100-m-90-iso8859-1 &
```

to get a 14pt font with all of the characters needed for ISO8859-1 (Latin 1, handling most of the languages of Western Europe, and many others, such as Hawaiian, Indonesian, and Swahili).

Your system manager may be able to tell you about additional window system fonts that may also be available, but are not loaded by default. For example, at the maintainer's site, there is a large collection of Asian and European fonts installed in the **emacs**(1) editor tree. To add, say, the European collection, in a shell window type

```
xset fp+ /usr/local/share/emacs/fonts/European
xset fp rehash
```

The new fonts will then be available, and will be listable by **xlsfonts**(1). You can make those additions permanent by adding those two commands to your *$HOME/.xinitrc* or *$HOME/.xsession* file; the name is platform-dependent, so the best choice is to make them identical, with one a symbolic link to the other.

Use

```
xset q
```

to find out what font directories are currently in the font search path.

Each X Window System font directory has a `fonts.dir` text file that maps short file names to long font names. There is sometimes also a `fonts.alias` text file to provide short aliases for the otherwise rather long font names used in the X Window System. You can scan those files to see what is available.

Recent versions of **xterm**(1) have a special option, **−u8**, to handle UTF-8 multibyte encoding, but you then need to use a font with the corresponding character repertoire:

```
xterm -u8 -fn \
    -misc-fixed-medium-r-normal--20-200-75-75-c-100-iso10646-1 &
```

Documentation for hoc in other languages

Internationalized documentation will usually augment, rather than replace, the English documentation. That way, translations can be developed incrementally. Thus, in a French environment, **help( )** responds in English, while output from **aide( )** is in French. On startup, **hoc** will then usually display a greeting in two languages: English, and the local one. Here is what this looks like in the French locale:

```
% env LANG=fr hoc
------------------------------------------------------------
Welcome to the extensible high-order calculator, hoc.
This is hoc version 7.0.0.beta [15-Dec-2001].
Type help() for help, news() for news, and author() for author
information.
This system supports IEEE 754 floating-point arithmetic.
------------------------------------------------------------
------------------------------------------------------------
Bienvenue à la calculatrice, hoc.
C'est la version 7.0 du 15 décembre 2001.
Taper aide() pour de l'assistance, nouvelles() pour des
nouvelles, et auteur() pour des renseignements sur les
auteurs.
Cet ordinateur supporte l'arithmétique en virgule flottante du
standard IEEE 754.
------------------------------------------------------------
```

The maintainer will be grateful for contributions of additional translations of **hoc** help files and internal messages!

HOC SUPPORT IN GNU EMACS

When **hoc** is installed properly, it adds a new library, *hoc.el*, to the *emacs/site-lisp* directory, which should always be included in the **emacs**(1) *load-path* variable (in an editor session, type `C-h vload-path` to display it).

By suitable manual edits to the *site-init.el* file in that directory, your system manager could make **hoc-mode** support automatically available, but the **hoc** installation process cannot safely do that automatically.

You can test whether this has been done at your site by visiting a new file with extension *.hoc*; if the **emacs**(1) mode line shows `(hoc ...)`, instead of something else, like `(fundamental ...)`, then you need do nothing more: **hoc-mode** is already fully installed.

Otherwise, in order to avoid the need for tedious manual loading of the **hoc** support in **emacs**(1), add this snippet of Emacs Lisp code at the end of your *$HOME/.emacs* initialization file:

```
(if (string-lessp (substring emacs-version 0 2) "19") ; earlier than 19.x
    (progn
      (setq auto-mode-alist
            (cons (cons "\.hoc$" 'hoc-mode) auto-mode-alist))
      (autoload 'hoc-mode "hoc"
        "Enter hoc mode." t nil))
  (progn
    (if (not (assoc "\.hoc$" auto-mode-alist))
        (setq auto-mode-alist
              (cons (cons "\.hoc$" 'hoc-mode) auto-mode-alist)))
    (autoload 'hoc-mode "hoc"
      "Enter hoc (high-order calculator) mode." t nil)))
```

There are two sections in this code, one for (now very old) **emacs**(1) versions before 19.x, and the other for all newer versions. They add a binding between files with extension *.hoc* and **hoc-mode** in **emacs**(1), and arrange for the *hoc.el* library to be loaded the first time that it is required.

DESCRIPTIONS OF BUILT-IN FUNCTIONS AND PROCEDURES

These descriptions are taken from the output of the corresponding **help_xxx( )** functions, and, apart from font differences, are intended to be identical to them. The **help_xxx( )** functions are considered to be the definitive documentation of each function.

In the following descriptions, square brackets on number ranges indicate that the endpoint is *included*; parentheses indicate that the endpoint is *excluded*.

abort(message) — **abort(message)** prints **message**, then aborts evaluation of the current expression, returning to top-level without further processing of the remainder of the current statement or function/procedure call chain. The message should include the name of the function calling **abort()**, since there is currently no function call traceback, and end with a newline.

abs(x) — **abs(x)** returns the absolute value of **x**.

acos(x) — **acos(x)** returns the arc cosine of **x**. **x** must be in [−1...+1].

acos(x) — **acos(x)** returns the arc cosine of **x**. **x** must be in [−1...+1].

acosh(x) — **acosh(x)** returns the inverse hyperbolic cosine of **x**. **x** must be outside the interval (−1...+1).

asinh(x) — **asinh(x)** returns the inverse hyperbolic sine of **x**.

atan(x) — **atan(x)** returns the arc tangent of **x**.

atanh(x) — **atanh(x)** returns the inverse hyperbolic tangent of **x**.

author() — **author()** prints information about the program authors.

cbrt(x) — **cbrt(x)** returns the cube root of **x**.

ceil(x) **ceil(x)** returns the smallest integer greater than or equal to **x**.

copysign(x,y) **copysign(x,y)** returns a value with the magnitude of **x**, and the sign of **y**.

cos(x) **cos(x)** returns the cosine of **x** (**x** in radians). Expect severe accuracy loss for large |**x**|.

cosd(x) **cosd(x)** returns the cosine of **x** (**x** in degrees). Expect severe accuracy loss for large |**x**|.

cosh(x) **cosh(x)** returns the hyperbolic cosine of **x**.

cpulimit(t) **cpulimit(t)** sets the CPU time limit from now to an additional **t** seconds, sets the system variable **__CPU_LIMIT__** to **t**, and returns the current CPU time limit, which is always measured from the *start* of the job.

If the limit is exceeded, execution of the current expression is aborted, control returns to the top-level interpreter, and the time limit is incremented by the current value of **__CPU_LIMIT__**.

Although **t** may be fractional, on most operating systems, the time limit is an integer, so **t** will be rounded up internally to the nearest integer before setting the time limit.

If resource usage and limits are not supported on the current platform, this function has no effect, other than setting **__CPU_LIMIT__**, and returning Infinity.

By default, there is no time limit for the job (although some operating systems may impose such limits).

Negative, zero, and NaN arguments are treated like Infinity.

NB: This function is *experimental*, and may be withdrawn in future versions.

erf(x) **erf(x)** returns the error function of **x**.

erfc(x) **erfc(x)** returns the complementary error function of **x**.

errbits(x,y) **errbits(x,y)**, with **y** an approximation to **x**, returns the number of bits that **y** is in error by.

eval(string) **eval(string)** pushes its argument string, which must contain valid **hoc** code, onto the input stack so that it will be evaluated next.

This function makes it possible for **hoc** programs to construct new **hoc** code on-the-fly and then run it.

There is a limit, set by the compiled-in dimension of the input pushback buffer, on the size of the expression that can be evaluated, but it is fairly large: it should be at least 10K characters in all **hoc** implementations. In this implementation, it is set to *nnn*. [Print the value of **__MAX_PUSHBACK__** in your implementation.]

exp(x) **exp(x)** returns the exponential function of **x**, **E^x**.

expm1(x) **expm1(x)** returns the exponential function of **x**, less 1: **E^x − 1**.

For small **x**, **exp(x)** is approximately 1, so there is serious subtraction loss in directly using **exp(x) − 1**; **expm1(x)** avoids this loss.

From Sun Solaris documentation: ''*The* **expm1( )** *and* **log1p( )** *functions are useful for financial calculations of* **((1 + x)^n − 1) / x**, *namely:*

expm1(n * log1p(x))/x

when **x** *is very small (for example, when performing calculations with a small daily interest rate). These functions also simplify writing accurate inverse hyperbolic functions.*''

exponent(x) **exponent(x)** returns the base-2 exponent of **x**, such that

x == significand(x) * 2^exponent(x)

where |**significand(x)**| is in [1...2).

For IEEE 754 arithmetic, normal numbers have **exponent(x)** in [-1022...1023] and subnormal numbers, if supported, have **exponent(x)** in [-1074...1023].

WARNING: The power **2^exponent(x)** will underflow to zero for IEEE 754 subnormal numbers, so for such numbers, the right-hand side must be computed with suitable scaling, like this:

(significand(x) * 2^(exponent(x) + 52)) * 2^(-52)

factorial(n) **factorial(n)** returns **n! = n*(n–1)*(n–2)*...*1**, where **1! == 0! == 1**, by definition. Negative arguments generate a call to **abort( )**.

floor(x) **floor(x)** returns the greatest integer less than or equal to **x**.

fmod(x,y) **fmod(x,y)** returns the remainder of the division of **x** by **y**.

gamma(x) **gamma(x)** returns the Gamma (generalized factorial) function of **x**.

gcd(x,y) **gcd(x,y)** returns the greatest common divisor of **x** and **y**.

getenv(envvar) **getenv(envvar)** returns the string value of the environment variable **envvar**, or an empty string if it is not defined.

hexfp(x) **hexfp(x)** returns a string containing the hexadecimal floating-point representation of **x**, in the form

"+0x1.hhhhh...p+ddddd"

Trailing zeros in the fraction, and leading zeros in the exponent, are dropped, and the sign is always included.

See also **help_hexint()**, **help_number()**, and **help_string()**.

hexint(x) **hexint(x)** returns a string containing the hexadecimal integer representation of **x**, if that is possible, in the form

"+0xhhhhh..."

Leading zeros are dropped, and the sign is always included.

If **x** is too big to represent as an exact integer, then the floating-point representation, **hexfp(x)**, is returned instead.

See also **help_hexfp()**, **help_number()**, and **help_string()**.

hypot(x,y) **hypot(x,y)** function computes the length of the hypotenuse of a right-angled triangle, **sqrt(x^2 + y^2)**, but without accuracy loss or range limitation from premature overflow or underflow.

This function has possibly unexpected behavior for exceptional arguments: when either argument is Infinity, then the result is Infinity, *even if* the other argument is a NaN! The explanation is found on the 4.3BSD manual page:

> ... programmers on machines other than a VAX (it has no infinity) might be surprised at first to discover that **hypot(+infinity,NaN) = +infinity**. This is intentional; it happens because **hypot(infinity,v) = +infinity** for all **v**, finite or infinite. Hence **hypot(infinity,v)** is independent of **v**. Unlike the reserved operand on a VAX, the IEEE NaN is designed to disappear when it turns out to be irrelevant, as it does in **hypot(infinity,NaN)**. ...

ilogb(x) **ilogb(x)** returns the exponent part of **x**, that is, **int(log2(x))**.

index(s,t) **index(s,t)** returns the index of string **t** in string **s**, counting from 1, or 0 if **t** is not found in **s**.

int(x) **int(x)** returns the integer part (truncated toward zero) of **x**.

isfinite(x) **isfinite(x)** returns 1 (true) if **x** is finite and otherwise, 0 (false).

isinf(x) **isinf(x)** returns 1 (true) if **x** is Infinite, and otherwise, 0 (false).

isnan(x) **isnan(x)** returns 1 (true) if **x** is a NaN, and otherwise, 0 (false).

isnormal(x) **isnormal(x)** returns 1 (true) if **x** is finite and normalized and not subnormal, and otherwise, 0 (false).

isqnan(x) **isqnan(x)** returns 1 (true) if **x** is a quiet NaN, and otherwise, 0 (false).

On some architectures (e.g., Intel x86 and MIPS), there is only one type of NaN. **isqnan(x)** is then defined to return **isnan(x)**.

issnan(x) **issnan(x)** returns 1 (true) if **x** is a signaling NaN, and otherwise, 0 (false).

On some architectures (e.g., Intel x86 and MIPS), there is only one type of NaN. **issnan(x)** is then defined to return **isnan(x)**.

You can test whether your system has both quiet and signaling NaNs like this: **issnan(NaN)**. The result is 0 (false) if distinct NaN types are available, and 1 (true) if not.

issubnormal(x) **issubnormal(x)** returns 1 (true) if **x** is subnormal (formerly, denormalized), and otherwise, 0 (false).

J0(x) **J0(x)** returns the Bessel function of the first kind of order 0 of **x**.

J1(x) **J1(x)** returns the Bessel function of the first kind of order 1 of **x**.

Jn(n,x) **Jn(n,x)** returns the Bessel function of the first kind of integral order **n** of **x**.

lcm(x,y) **lcm(x,y)** returns the least common multiple of **int(x)** and **int(y)**.

ldexp(x,y) **ldexp(x,y)** returns **x * 2^(int(y))**.

lgamma(x) **lgamma(x)** returns the natural logarithm of **gamma(x)**.

Because **gamma(x)** has poles at zero and at negative integer values, and grows factorially with increasing **x**, it reaches the floating-point overflow limit fairly quickly. For 64-bit IEEE 754 arithmetic, this happens at approximately **x = 206.779**. However, **lgamma(x)** is representable almost to the overflow limit. In 64-bit IEEE 754 arithmetic, this happens at approximately **x = 2.55e+306** (the overflow limit is 1.80e+308).

Unfortunately, there is mathematically-unavoidable accuracy loss when **gamma(x)** is computed from **exp(lgamma(x))**, so you should avoid the logarithmic form unless you really need large arguments that would cause overflow.

ln(x) **ln(x)** returns the natural (base-E) logarithm of **x**.

load(filename) **load(filename)** reads input from the specified file. The file can be prepared by hand, or by the **save( )** command.

Loaded files can themselves contain **load( )** commands, with nesting up to some unknown limit imposed by the host operating system on the maximum number of simultaneously-open files for a process, user, or the entire system.

This command can be disabled for security reasons by the command-line **−noload** option.

The return value is an empty string on success, and otherwise, an error message.

log(x) **log(x)** returns the natural (base-E) logarithm of **x**.

log10(x) **log10(x)** returns the logarithm to the base 10 of **x**.

log1p(x) **log1p(x)** returns **log(1 + x)**, but without accuracy loss for small |**x**|. **x** must be in (−1. . .infinity].

log2(x) **log2(x)** returns the logarithm to the base 2 of **x**.

logfile(filename) **logfile(filename)** logs the session on the specified file, which, for security reasons, *must* be a new file. It is a normal text which you can edit, print, and view.

Input is recorded verbatim. Output is recorded in comments. This permits the logfile to be read by **hoc** later, allowing a session to be replayed.

If a logfile is already opened, it is closed before opening the new one.

Logging may be turned on and off with **logon( )** and **logoff( )**, and can be entirely disabled for security reasons by the command-line **−no-logfile** option.

The return value is an empty string on success, and otherwise, an error message.

logoff( ) **logoff( )** suspends logging to any open log file. It is *not* an error if there is no current log file.

logon( ) **logon( )** restores logging to any open log file. It is *not* an error if there is no current log file.

macheps(x) **macheps(x)** returns the generalized machine epsilon of **x**, the smallest number which, when added to **x**, produces a sum that still differs from **x**: **(x + macheps(x)) != x**.

macheps(1) is the normal machine epsilon.

macheps(−x) is **macheps(x)/base**, or equivalently, the smallest number that can be subtracted from **x** with the result still different from **x**.

macheps(0) is the smallest representable floating-point number. Depending on the host system, it may be a normal number, or a subnormal number (invoke **help_subnormal( )** for details).

max(x,y) **max(x,y)** returns the larger of **x** and **y**.

If *either* argument is a NaN, the result is a NaN.

maxnormal( ) **maxnormal( )** returns the maximum positive normal number.

min(x,y) **min(x,y)** returns the smaller of **x** and **y.**

If *either* argument is a NaN, the result is a NaN.

minnormal( ) **minnormal( )** returns the minimum positive normal number.

minsubnormal( ) **minsubnormal( )** returns the minimum positive subnormal number. If subnormals are not supported, then it returns the minimum normal number instead.

nearest(x,y) **nearest(x,y)** returns the next different machine number nearest **x**, in the direction of the infinity with the same sign as **y.**

nextafter(x,y) **nextafter(x,y)** returns the nearest machine number nearest **x**, in the direction of the infinity with the same sign as **y**.

nint(x) **nint(x)** returns the nearest integer to **x**, rounding away from zero in case of a tie.

now( ) **now( )** returns the current date, in the form "Dec 8 2001". If the month day has only one digit, then it is preceded by an extra space, so that the format is uniformly "MMM DD YYYY".

number(s) **number(s)** converts the string **s** to a number and returns it.

s should contain either a hexadecimal floating-point number, a hexadecimal integer, a decimal floating-point number, a decimal integer, or a representation of NaN or Infinity.

If **s** contains a number followed by unrecognizable text, the number is converted and returned, and the following text is silently ignored. Otherwise, the return value is 0, and the text is silently ignored. Thus, **number("123abc")** returns 123, and **number("abc")** returns 0.

This function is an inverse of **hexfp()**, **hexint()**, and **string()**:

number(hexfp(x)) == **x** [for all numeric **x**]
number(hexint(x)) == **x** [for all numeric **x**]
number(string(x)) == **x** [for all numeric **x**]

See also **help_hexint()**, **help_hexfp()**, and **help_string()**.

printenv(prefix) **printenv(prefix)** prints the names and values of all environment variables whose names match that **prefix**. Use **printenv("")** to match all names.

putenv(envvar,newval)

putenv(envvar,newval) replaces the current string value of the environment variable **envvar** with **newval**, and returns its old value.

This affects subsequent calls to **getenv()**, but does *not* affect the environment of the parent process.

You can use this function to set locale environment variables that control the output of dates and times, in order to get internationalized output from **strftime()**.

rand( ) **rand( )** returns a pseudo-random number uniformly distributed on (0...1). Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

See **help_randint( )** for uniformly-distributed integers in an interval, and **help_randl( )** for logarithmically-distributed pseudo-random numbers.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

randint(x,y) **randint(x,y)** returns a pseudo-random integer uniformly distributed on **[int(x)...int(y)]**. Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

randl(x) **randl(x)** returns a pseudo-random number logarithmically distributed on **(1,exp(x))**. Unless the seed is changed (see **help_setrand( )**), successive runs of the same program will generate the same sequence of pseudo-random numbers.

This function can be used to generate logarithmic distributions on any interval: **a*randl(ln(b/a))** is logarithmically distributed on **(a...b)**.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

remainder(x,y) **remainder(x,y)** returns the remainder **r = x − n*y**, where **n** is the integral value nearest the exact value **x/y**. When **|n − x/y| = 1/2**, the value of **n** is chosen to be even.

rint(x) **rint(x)** returns the integral value nearest **x** in the direction of the current IEEE 754 rounding mode.

rsqrt(x) **rsqrt(x)** returns the reciprocal square root, **1/sqrt(x)**.

save(filename,prefix) **save(filename,prefix)** saves the state of the current session in the specified file, which, for security reasons, *must* be a new file.

If the prefix string is not empty, then only symbols whose initial characters match the prefix string are saved.

Symbols are output in strict alphabetical order

Reserved symbol names (those beginning with two or more underscores) are not saved. Predefined immutable names are also excluded.

The saved file is a normal text file that can be later read by **hoc** on any platform.

[NB: A temporary implementation restriction also excludes user-defined immutable names, and all functions and procedures.]

This command can be disabled for security reasons by the command-line **−nosave** option.

The return value is an empty string on success, and otherwise, an error message.

scalb(x,y) **scalb(x,y)** returns **x * 2^(int(y))**.

second( ) **second( )** returns the CPU time in job seconds since some fixed time in the past. Take the difference of two bracketing calls to get the elapsed CPU time for a block of code. For example,

```
PREC = 3
x = 1
t = second( )
for (k = 1; k < 1000000; ++k) x *= 1
second( ) - t
4.73
```

setrand(x) **setrand(x)**, where **x** should be a large integer, sets the seed of the pseudo-random number generator to **x**, and returns the old seed.

As a special case, when **x** is zero, **x** is ignored, and a new seed is constructed from a random number multiplied by either the calendar time (if available), or the process number (if available), or the next pseudo-random number.

If **setrand(x)** is never called, then **rand( )**, **randint( )**, and **randl(x)** will each return the same sequence of pseudo-random numbers: see **help_rand( )**, **help_randint( )**, and **help_randl( )**.

The pseudo-random generator algorithm is platform-independent, allowing reproduction of the same number sequence on any computer architecture.

significand(x) **significand(x)** returns the significand of **x**, **s,** such that **x = s * 2^n**, with **s** in [1,2), and **n** an integer.

See **help_exponent()** for how to extract the exponent, **n**.

sin(x) **sin(x)** returns the sin of **x** (**x** in radians). Expect severe accuracy loss for large |**x**|.

sind(x) **sind(x)** returns the sin of **x** (**x** in degrees). Expect severe accuracy loss for large |**x**|.

sinh(x) **sinh(x)** returns the hyperbolic sin of **x**.

sqrt(x) **sqrt(x)** returns the square root of **x**. **x** must be in [−0. . .Infinity].

Special case: **sqrt(−0)** → **−0**.

strftime(format,time) **strftime(format,time)** converts a numeric time measured in seconds since the epoch (usually obtained from **systime( )**) to a formatted string determined by one

or more of these format items:

- **%A** the locale's full weekday name.
- **%a** the locale's abbreviated weekday name.
- **%B** the locale's full month name.
- **%b** the locale's abbreviated month name.
- **%c** the locale's appropriate date and time representation.
- **%d** the day of the month as a decimal number (01–31).
- **%H** the hour (24-hour clock) as a decimal number (00–23).
- **%I** the hour (12-hour clock) as a decimal number (01–12).
- **%j** the day of the year as a decimal number (001–366).
- **%M** the minute as a decimal number (00–59).
- **%m** the month as a decimal number (01–12).
- **%p** the locale's equivalent of either ''AM'' or ''PM''.
- **%S** the second as a decimal number (00–60).
- **%U** the week number of the year (Sunday as the first day of the week) as a decimal number (00–53).
- **%W** the week number of the year (Monday as the first day of the week) as a decimal number (00–53).
- **%w** the weekday (Sunday as the first day of the week) as a decimal number (0–6).
- **%X** the locale's appropriate time representation.
- **%x** the locale's appropriate date representation.
- **%Y** the year with century as a decimal number.
- **%y** the year without century as a decimal number (00–99).
- **%Z** the time zone name.
- **%%** is replaced by '%'.

string(x) **string(x)** returns a string containing the decimal representation of **x**, either in integer form (if **x** is exactly representable that way), or in floating-point form.

See also **help_hexfp()**, **help_hexint()**, and **help_number()**.

substr(s,start,len) **substr(s,start,len)** returns a substring of string **s** beginning at character **start** (counting from 1), of length at most **len**. If **start** is outside the string, it is moved to the nearest endpoint, *without* adjusting **len**. Fewer than **len** characters will be returned if the substring extends outside the original string.

systime( ) **systime( )** returns the calendar time in seconds since the epoch. On UNIX systems, the epoch starts on January 1, 1970 00:00:00 UTC. Other operating systems make different choices. It can be converted to a formatted time string with **strftime( )**.

tan(x) **tan(x)** returns the tangent of **x** (**x** in radians). Expect severe accuracy loss for large |**x**|.

tand(x) **tand(x)** returns the tangent of **x** (**x** in degrees). Expect severe accuracy loss for large |**x**|.

tanh(x) **tanh(x)** returns the hyperbolic tangent of **x**.

tolower(s) **tolower(s)** returns a copy of string **s** with uppercase letters converted to lowercase, and all other characters unchanged.

Which characters are considered uppercase depends on the locale. On UNIX, this is determined by the **LC_CTYPE** environment variable.

toupper(s) **toupper(s)** returns a copy of string **s** with lowercase letters converted to uppercase, and all other characters unchanged.

Which characters are considered lowercase depends on the locale. On UNIX, this is determined by the **LC_CTYPE** environment variable.

trunc(x) **trunc(x)** returns the integer part of **x**, with the fractional part discarded.

who(prefix) **who(prefix)** prints all symbols whose initial characters match the **prefix** string, grouped by category. To print all symbols, use an empty prefix: **who("")**.

Y0(x) **Y0(x)** returns the Bessel function of the second kind of order 0 of **x**, for **x >= 0**. This function is also called *Weber's function*.

Y1(x) **Y1(x)** returns the Bessel function of the second kind of order 1 of **x**, for **x >= 0**. This function is also called *Weber's function*.

Yn(n,x) **Yn(n,x)** returns the Bessel function of the second kind of integral order **n** of **x**, for **x >= 0**. This function is also called *Weber's function*.

IMPLEMENTATION LIMITS

hoc has a few compile-time dimensions that limit the size of certain objects. Their current values are available in these predefined immutable constants:

__MAX_NAME__ Longest identifier name.

__MAX_PUSHBACK__ Input pushback buffer size, and thus, also the limit on the length of the string argument that the **eval( )** function can handle.

__MAX_STRING__ Longest character string constant.

__MAX_TOKEN__ Longest numeric token.

The function **help_limits()** can be conveniently used to display their current values.

A design goal for future versions of **hoc** is to eliminate these limits entirely, making them subject only to available memory.

EXAMPLES

```
func gcd( ) {
        ## gcd(i,j) returns the greatest common denominator of i and j
        temp = abs($1) % abs($2)
        if(temp == 0) return abs($2)
        return gcd($2, temp)
}
for(i=1; i<12; i++) print gcd(i,12)
print "\n"
1 2 3 4 1 6 1 4 3 2 1
### Print a table of the representable negative powers of 2
k = 0
x = 1
while (x > 0) \
{
        print "2^(", k, ") = ", x, "\n"
        k--
        x /= 2
}
```

```
2^(0 ) = 1
2^(-1 ) = 0.5
2^(-2 ) = 0.25
2^(-3 ) = 0.125
...
2^(-1072 ) = 1.9762625833649862e-323
2^(-1073 ) = 9.8813129168249309e-324
2^(-1074 ) = 4.9406564584124654e-324
```

INITIALIZATION FILES

On startup, after processing any command-line options that suppress initialization files, **hoc** checks for the existence of local system-wide initialization files,

- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/hoc.rc`,
- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/LN/hoc.rc`,
- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/help.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/LN/help.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/translations.hoc`,
- `/usr/local/share/lib/hoc/hoc-7.0.0.beta/LN/translations.hoc`,

(LN is replaced by the locale name (see the **INTERNATIONALIZATION** section above), if one is defined, and otherwise, that file is omitted), and a private initialization file,

- `$HOME/.hocrc`,

in that order. Any that exist are automatically processed before the remaining command-line options are handled.

This feature allows for local customization of **hoc**, usually for additional constants and functions, as well as for locale-specific translations of output strings.

In initialization files, the **load( )**, **logfile( )**, and **save( )** commands are *always* available, even if command-line options disable them from use later in the job.

If GNU *readline* library support is available in **hoc**, then its initialization file, `$HOME/.inputrc`, (overriddable by setting an alternate filename in the value of the **INPUTRC** environment variable), can be used for customization of key bindings for command completion, editing, and recall. To restrict any such bindings to **hoc**, put them in a conditional like this:

```
$if hoc
    ...
$endif
```

SEE ALSO

awk(1), **bc**(1), **dc**(1), **dircolors**(1), **emacs**(1), **expr**(1), **genius**(1), **locale**(1), **vi**(1), **xlsfonts**(1), **xterm**(1).
Brian W. Kernighan and Rob Pike, *The UNIX Programming Environment*, Prentice-Hall, 1984,
ISBN 0-13-937699-2 (hardcover), 0-13-937681-X (paperback),
LCCN: QA76.76.O63 K48 1984.

BUGS

All components of a **for** statement must be non-empty.

Error recovery is imperfect within function and procedure definitions.

The treatment of newlines is not exactly user-friendly.

Arguments `$1`, etc., are not really variables and thus won't work in constructs like, for instance, `$1++`.

Functions and procedures typically have to be declared before use, which makes mutual recursion at first sight impossible. The workaround is to first define a dummy version of one of them. For example, here is an unusual implementation of a pair of functions, each of which returns the factorial of its argument:

```
func foo() return 0
func bar() {if ($1 > 0) return $1 * foo($1-1) else return 1}
```

```
func foo() {if ($1 > 0) return $1 * bar($1-1) else return 1}
```

AVAILABILITY

hoc is highly portable, and vastly smaller than a compiler for a major programming language, so it should be usable on all computing platforms. When a C or C++ compiler is available, **hoc** can be easily built, validated, and installed using the distribution source code from its master archive:

```
ftp://ftp.math.utah.edu/pub/hoc
http://www.math.utah.edu/pub/hoc/
```

For platforms where suitable compilers are often not installed, there may be binary distributions available at those locations.

COPYRIGHT

ACKNOWLEDGEMENTS

The **hoc** version 7 developer and maintainer (Nelson H. F. Beebe <beebe@math.utah.edu>) thanks the AT&T/Lucent Bell Labs people (current and former), notably Ken Thompson, Dennis Ritchie, Brian Kernighan, Rob Pike, John Bentley, Bill Plauger, Stu Feldman, David Gay, Norm Schryer, and Bjarne Stroustrup for developing the wonderful UNIX and C/C++ programming environment, and being a constant source of inspiration for software development and superb book authoring.

He also thanks the many people at the Free Software Foundation, for enriching UNIX with GNUware, and most notably, Richard Stallman for **emacs**(1) and **gcc**(1), for founding the FSF and the GNU Project, and for vigorous campaigning to keep software freely distributable.

Finally, he thanks friends and colleagues on the **hoc** help facility translation team for assistance in internationalization: Hugo Bertete-Aguirre (Portuguese), Andrej Cherkaev (Russian), Tanya Damjanovic (Serbian), Michel Debar (French), Miguel Dumett (Spanish), Henryk Hecht (Polish), Michael Hohn (German), Ismail Küçük (Turkish), Young Seon Lee (Korean), Dragan Milicic (Croatian), and Jingyi Zhu (Chinese). [The English and Danish, and part of the French, help facilities were written by the maintainer.]