Hoc - An Interactive Language For Floating Point Arithmetic

Brian Kernighan
Rob Pike

ABSTRACT

Hoc is a simple programmable interpreter for floating point expressions. It has C-style control flow, function definition and the usual numerical built-in functions such as cosine and logarithm.

1. Expressions

Hoc is an expression language, much like C: although there are several control-flow statements, most statements such as assignments are expressions whose value is disregarded. For example, the assignment operator = assigns the value of its right operand to its left operand, and yields the value, so multiple assignments work. The expression grammar is:

expr:	*number*
|	*variable*
|	*(expr)*
|	*expr binop expr*
|	*unop expr*
|	*function (arguments)*

Numbers are floating point. The input format is that recognized by *scanf*(3): digits, decimal point, digits, *e* or *E*, signed exponent. At least one digit or a decimal point must be present; the other components are optional.

Variable names are formed from a letter followed by a string of letters and numbers. *binop* refers to binary operators such as addition or logical comparison; *unop* refers to the two negation operators, '!' (logical negation, 'not') and '−' (arithmetic negation, sign change). Table 1 lists the operators.

Table 1: Operators, in decreasing order of precedence

`^`	exponentiation (FORTRAN **), right associative
`! -`	(unary) logical and arithmetic negation
`* /`	multiplication, division
`+ -`	addition, subtraction
`> >=`	relational operators: greater, greater or equal,
`< <=`	less, less or equal,
`== !=`	equal, not equal (all same precedence)
`&&`	logical AND (both operands always evaluated)
`||`	logical OR (both operands always evaluated)
`=`	assignment, right associative

Functions, as described later, may be defined by the user. Function arguments are expressions separated by commas. There are also a number of built-in functions, all of which take a single argument, described in Table 2.

Table 2: Built-in Functions

`abs(x)`	$\lvert x \rvert$, absolute value of x
`atan(x)`	arc tangent of x
`cos(x)`	$\cos(x)$, cosine of x
`exp(x)`	e^x, exponential of x
`int(x)`	integer part of x, truncated towards zero
`log(x)`	$\log(x)$, logarithm base e of x
`log10(x)`	$\log_{10}(x)$, logarithm base 10 of x
`sin(x)`	$\sin(x)$, sine of x
`sqrt(x)`	$\sqrt{x}$, $x^{\frac{1}{2}}$

Logical expressions have value 1.0 (true) and 0.0 (false). As in C, any non-zero value is taken to be true. As is always the case with floating point numbers, equality comparisons are inherently suspect.

Hoc also has a few built-in constants, shown in Table 3.

Table 3: Built-in Constants

`DEG`	57.29577951308232087680	$180/\pi$, degrees per radian
`E`	2.71828182845904523536	e, base of natural logarithms
`GAMMA`	0.57721566490153286060	γ, Euler-Mascheroni constant
`PHI`	1.61803398874989484820	$(\sqrt{5}+1)/2$, the golden ratio
`PI`	3.14159265358979323846	π, circular transcendental number

2. Statements and Control Flow

Hoc statements have the following grammar:

stmt: *expr*
  | *variable = expr*
  | *procedure (arglist)*
  | *while (expr) stmt*
  | *if (expr) stmt*
  | *if (expr) stmt else stmt*
  | *{ stmtlist }*
  | *print expr-list*
  | *return optional-expr*

stmtlist: (nothing)
  | *stmlist stmt*

An assignment is parsed by default as a statement rather than an expression, so assignments typed interactively do not print their value.

Note that semicolons are not special to *hoc*: statements are terminated by newlines. This causes some peculiar behavior. The following are legal *if* statements:

```
if (x < 0) print(y) else print(z)

if (x < 0) {
    print(y)
} else {
    print(z)
}
```

In the second example, the braces are mandatory: the newline after the *if* would terminate the statement and produce a syntax error were the brace omitted.

The syntax and semantics of *hoc* control flow facilities are basically the same as in C. The *while* and *if* statements are just as in C, except there are no *break* or *continue* statements.

3. Input and Output: *read* and *print*

The input function *read*, like the other built-ins, takes a single argument. Unlike the built-ins, though, the argument is not an expression: it is the name of a variable. The next number (as defined above) is read from the standard input and assigned to the named variable. The return value of *read* is 1 (true) if a value was read, and 0 (false) if *read* encountered end of file or an error.

Output is generated with the *print* statement. The arguments to *print* are a comma-separated list of expressions and strings in double quotes, as in C. Newlines must be supplied; they are never provided automatically by *print*.

Note that *read* is a special built-in function, and therefore takes a single parenthesized argument, while *print* is a statement that takes a comma-separated, unparenthesized list:

```
while (read(x)) {
     print "value is ", x, "\n"
}
```

4. Functions and Procedures

Functions and procedures are distinct in *hoc*, although they are defined by the same mechanism. This distinction is simply for run-time error checking: it is an error for a procedure to return a value, and for a function *not* to return one.

The definition syntax is:

function: *func name() stmt*

procedure: *proc name() stmt*

name may be the name of any variable — built-in functions are excluded. The definition, up to the opening brace or statement, must be on one line, as with the *if* statements above.

Unlike C, the body of a function or procedure may be any statement, not necessarily a compound (brace-enclosed) statement. Since semicolons have no meaning in *hoc*, a null procedure body is formed by an empty pair of braces.

Functions and procedures may take arguments, separated by commas, when invoked. Arguments are referred to as in the shell: *$3* refers to the third (1-indexed) argument. They are passed by value and within functions are semantically equivalent to variables. It is an error to refer to an argument numbered greater than the number of arguments passed to the routine. The error checking is done dynamically, however, so a routine may have variable numbers of arguments if initial arguments affect the number of arguments to be referenced (as in C's *printf*).

Functions and procedures may recurse, but the stack has limited depth (about a hundred calls). The following shows a *hoc* definition of Ackermann's function:

```
$ hoc
func ack() {
        if ($1 == 0) return $2+1
        if ($2 == 0) return ack($1-1, 1)
        return ack($1-1, ack($1, $2-1))
}
ack(3, 2)
        29
ack(3, 3)
        61
ack(3, 4)
hoc: stack too deep near line 8
...
```

5. Examples

Stirling's formula:

$$n! \sim \sqrt{2n\pi}(n/e)^n(1 + \frac{1}{12n})$$

```
$ hoc
func stirl() {
    return sqrt(2*$1*PI) * ($1/E)^$1*(1 + 1/(12*$1))
}
stirl(10)
        3628684.7
stirl(20)
        2.4328818e+18
```

Factorial function, $n!$:

```
func fac() if ($1 <= 0) return 1 else return $1 * fac($1-1)
```

Ratio of factorial to Stirling approximation:

```
i = 9
while ((i = i+1) <= 20) {
        print i, "  ", fac(i)/stirl(i), "\n"
}
10   1.0000318
11   1.0000265
12   1.0000224
13   1.0000192
14   1.0000166
15   1.0000146
16   1.0000128
17   1.0000114
18   1.0000102
19   1.0000092
20   1.0000083
```