Why Must $0^0 = 1$?

W. Kahan
Univ. of Calif. @ Berkeley
Mar. 30, 1983

For typographical reasons, let us write X^Y in place of X^Y . Programs to compute this function exist in the run-time support library of many computer systems, but those programs tend to disagree about what to do with 0^0 and even (-3)^3 , sometimes producing unreasonable values like 0 and +27 resp., sometimes reasonable values like 1 and -27 , sometimes warning messages, and sometimes a mixture of the three. The purpose of this note is to provide a rationale for treating the case x^i uniformly when x is an integer or real variable and i is an integer variable. (The case x^y when the exponent y is real will be treated elsewhere [ ].) We shall cope smoothly with 0^0 , 0^(-1) and similar special cases in the numerical environment provided by the proposed IEEE standards p754 and p854 for floating-point arithmetic, in which provisions exist for Infinity and a symbol that is Not a Number (NaN); see [ ] and [ ].

Notation: As is conventional in FORTRAN, the letters i, j, k, l, m and n are reserved for integer variables; we shall further restrict m and n to positive integers.

Let R denote the finite real numbers, and R' the nonzero finite reals. We shall represent Infinity by the symbol ∞ ignoring its sign until later, when it will matter. Next let X denote that extension of the real numbers that includes Infinity; that is

$$X = R \cup \{\infty\} \quad \text{and} \quad R = \{0\} \cup R' .$$

The letters a, b, c, ..., g, h are reserved for finite reals (R) and the letters p, q, r, ..., x, y, z for extended real (X) variables. Finally, the symbol NaN stands for "Not a Number", and is the only thing in our universe that violates the equation

$$x = x .$$

Below are the Multiplication and Division tables for extended reals. (Addition and Subtraction are not needed for this note, and therefore not provided.) Certain side-effects are indicated by asterisks * ; the instances marked NaN * also signal an "Invalid Operation". The instance marked ∞ * also signals a "Divide by Zero" , but this really means "Infinity created precisely from finite operands".

MULTIPLICATION x y

	y = 0	y in R'	y = ⋈	y is NaN
x = 0	0	0	NaN *	y
x in R'	0	x y	⋈	y
x = ⋈	NaN *	⋈	⋈	y
x is NaN	x	x	x	x or y

DIVISION x/y

	y = ⋈	y in R'	y = 0	y is NaN
x = 0	0	0	NaN *	y
x in R'	0	x/y	⋈ *	y
x = ⋈	NaN *	⋈	⋈	y
x is NaN	x	x	x	x or y

Exponentiation: Descartes' definition

```
                   n
     c^n   =   c     =   c  c  c  c  ......  c  c  c  c
                             <-  n copies of   c   ->
```

is accepted universally only when c is a finite real number and n a positive integer. Our objective is to extend this definition so that x^i will be an extended real number for all extended reals x and every integer i ; furthermore, the extension must be parsimonious, entailing a minimum of assumptions and therefore implying a minimum of special cases. The definition that results from this parsimony is displayed in the table below, following which is an explanation to justify the entries therein that differ from other opinions.

EXPONENTIATION x^i

	i < 0	i = 0	i > 0
x in X	(1/x)^(-i)	1	x^i
x is NaN	x	1	x

Four aspects of the foregoing table deserve explanation. First is the assignment x^0 = 1 regardless of whether x = 0 , x = ⋈

or x is NaN . Second are the side-effects. Implicit in the calculation of 0^i = (1/0)^(-i) = M^(-i) when i < 0 is the emission of a Divide by Zero signal. Similarly, when x^i has a finite nonzero magnitude too big or too tiny to represent within the range of floating-point numbers then that magnitude must be coerced to a representable value and an Overflow or Underflow signal emitted. Third is the trouble caused by roundoff, which obliges us to use unobvious algorithms to avoid it as much as possible. Fourth is the attention that must be paid to the signs that the proposed IEEE standard attaches to Zero and Infinity . These four aspacts are explained below.

A Parsimonious Set of Postulates. Historically, the meaning of c^(-n) when -n < 0 appears to have evolved from the observation that c^(m+n) = (c^m) (c^n) for all m > 0 and n > 0 , followed by the insistence that this equation be satisfied for all integers m and n regardless of sign. We shall follow this path now only to demonstrate that its logical conclusion is a dead end.

We postulate the existence of a function P(x, i) that takes values in X for all x in X and every i in I , and satisfies the defining relations

1: P(x, 1) = x , and
2: P(x, i+j) = P(x, i) P(x, j) unless this product is NaN .

We hope to deduce that P(x, i) = x^i . (To this end, relation 1 is needed to preclude that P(x, i) be independent of x or else perversely x^(ki) for some k fixed arbitrarily.) From these relations soon follow familiar inferences:

```
  P(x, n)  =  x^n  =  x x x ... x x x   for every  n > 0 .
                      <-  n times  ->

  P(x, i n)  =  P(P(x, i), n)  =  P(x, i)^n   for every  n > 0 .
```

For every integer i , positive or negative or zero, and for all nonzero finite b and c in R'
P(c, -i) = 1/P(c, i) = P(1/c, i) and P(c, 0) = 1 .

Therefore the two relations above do define P(x, i) = x^i either when i > 0 or when x lies in R' ; but the remaining cases
P(0, 0), P(M, 0), P(0, -n) and P(M, -n) when -n < 0
are left ambiguous. The clearly undesirable assignments
P(0, 0) = P(0, -n) = 0 and P(M, 0) = P(M, -n) = M
are no less compatible with relations 1 and 2 then are the more reasonable assignments for 0^0, 0^(-n), M^0 and M^(-n) respectively tabulated above. Since relations 1 and 2 do not define P(x, i) uniquely for every integer i and all x in X , other relations must be chosen to characterize x^i .

Two relations can be chosen that are just as simple as the two above but fully effective; they are

I: $P(x, 0) = 1$, and
II: $P(x, 1+i) = x\,P(x, i)$ unless the product is NaN .

Although the first of these defining relations may seem a self-serving postulate in a document purporting to draw it as a conclusion, these postulates are not unprecedented; they may be found in texts like L. E. Sigler's "Algebra" (p. 104) published in 1976 by Springer-Verlag, New York. No text contains a proof that these postulates are effective over the extended reals X so the proof is outlined here.

By induction, as in Sigler's text, we may prove for every integer i and all x in R' that $P(x, i) = x\^i$. Also easy to prove is that $P(0, n) = 0\^n$ and $P(\infty, n) = \infty\^n$ for positive integers n . The remaining cases are $P(0, -n)$ and $P(\infty, -n)$. Now, $P(0, -1)$ cannot be finite lest $1 = P(0, 1-1) = 0\,P(0,-1) = 0$; therefore $P(0, -1) = \infty = 0\^(-1)$. Similarly, $P(\infty, -1) = 0 = \infty\^(-1)$. And a similar argument combined with induction establishes for $n > 1$ that $P(0, -n) = \infty = 0\^(-n)$ and $P(\infty, -n) = 0 = \infty\^(-n)$. Q.E.D.

Why Should NaN^0 = 1 ? The only other plausible choice is NaN , and if we were to follow the pattern set by the other binary arithmetic operations (+, -, *, /, rem) in the proposed IEEE standard, all of which reproduce a NaN when it is combined with any other operand, then we might think that NaN is the analogous choice for NaN^0 . But that analogy does not apply to the exponential function $x\^i$ with integer exponent i because $x\^0 = 1$ for *all* x in X , whereas no such constant value results from the other operations; for example, $0/x = 0$ for all x in X *except* $x = 0$, or $x-x = 0$ for all *finite* x , etc. Therefore the value, if any, assigned to NaN^0 must be derived from first principles.

The principal service rendered by a NaN is to permit a deferred judgment about a variable whose value is either unavailable (e.g. an uninitialized variable) or the result of an invalid calculation (like $0/0$) from which no numerical result could be anything but misleading. Rather than abort a computation as soon as a NaN appears, we might prefer to persevere in the hope that whatever circumstances produced the NaN will later be seen to render that NaN irrelevant to a correct conclusion. For instance, the statement

if $x = 0$ or $y/x < 3$ then $z := 15$.

should set $z := 15$ whenever $x = 0$ even if $y = 0$ too, despite that y/x is then NaN , and regardless of whether the semantics of the programming language allows "short-circuit" evaluation of logical expressions. Another example is a function $f(x, y, z, t)$ which turns out to be completely independent of, say, x ; then $f(\mathrm{NaN}, y, z, t) = f(x, y, z, t)$ seems like the most reasonable conclusion, especially if the variable x never appears in the expression that defines f . This is the analogy that motivates the assignment P(NaN, 0) = P(x, 0) = 1 for all x in the case of P(x, i) = x^i . In this light, any other definition for NaN^0 seems more anomalous than helpful.

Side Effects: Implicit in the definition of x^i by the table above is the emission of "Divide by Zero" signals from expressions like 0^(-3) , just as from 1/0 , to mark the precise creation of ∞ from finite operands. The same result approximates overflows like 0.00000001^(-9999999) , but with an "Overflow" signal instead. Similarly, "Underflow" must be signaled when x^i underflows, and "Inexact" when it suffers from roundoff. On the other hand, spurious over/underflow signals should not be emitted; that is why, when -n< 0 , the table above defines x^(-n) = (1/x)^n instead of 1/(x^n) ; the latter expression would reverse the over/underflow signals. However, the latter expression is the more accurate in the face of roundoff when over/underflow does not occur. Therefore, a program that computes x^i in accordance with the table above, but would minimize the degradation due to roundoff, must be appreciably more complicated than the table.

The Sign of x^i : The proposed IEEE standards assign signs to both zero and Infinity. Whereas the sign of Infinity has an obvious interpretation, the sign of zero cannot be used to distinguish +0 from -0 by normal numerical means; on the contrary, the standard says that the Comparison operation must assert +0 = -0 . However, 1/(+0) = +∞ > 1/(-0) = -∞ . The seeming paradox evaporates when the sign of zero is recognized as one bit of information, stored in an auxiliary variable that is attached to any ordinary variable that takes the value 0 , which propagates through arithmetic operations in a way that matters only when a function discontinuous at zero is calculated. The rules governing that propagation make sense when zero can be regarded as an approximation to a very tiny quantity of known sign but unknown magnitude. If these rules save a programmer the bother of creating and manipulating an explicit auxiliary variable, then they have repaid the nearly negligible cost of their implementation in the standard; otherwise the programmer can insert the same branches in his code as if zero had no sign. The effect upon x^i of those propagation rules can be inferred from the table above and summarized as follows for all x in X :

If i is odd then the sign of x^i matches the sign of x ;
otherwise the sign of x^i is " + " .

This rule costs almost nothing to implement.

A Program for x^i : P(x, i) is a real function that delivers
~~~~~~~~~~~~~~~~~~~   x^i for any integer variable i and real
variable x . These variables are passed to P by value. Each
procedure OverflowFlag(), UndrflowFlag(), OverflowTrap() and
UndrflowTrap() returns its current status and then replaces it by
its argument; in other words, it performs a swap.

```
Begin  P(x, i):
  If  i < 0  then begin
             OvSav := OverflowFlag(.false.) ;
             UnSav := UndrflowFlag(.false.) ;
             OvTrpSav := OverflowTrap(.none.) ;
             UnTrpSav := UndrflowTrap(.none.) ;
             z := P(x, -i) ;
             OvSav := OverflowFlag(OvSav) ;
             UnSav := UndrflowFlag(UnSav) ;
             OvTrpSav := OverflowTrap(OvTrpSav) ;
             UnTrpSav := UndrflowTrap(UnTrpSav) ;
             If  OvSav or UnSav  then  return  P := P(1/x, -i)
                 else  return  P := 1/z
             end
      else begin
             j := i rem 2  ... = 1 if  i  is odd,  0  if even ;
             i := i-j ;
             If  j = 1  then  z := x  else  z := 1 ;
             y := x ;
             while  i > 0  do begin
                    i := i/2 ;
                    y := y*y ;
                    j := i rem 2 ;
                    i := i-j ;
                    If  j = 1  then  z := z*y
                    end ;
             return  P := z
             end
  end.
```

Except for what is necessary to diminish the effect of roundoff while not reversing the over/underflow signals, this code contains nothing that might not appear in an efficient code, one that nearly minimizes the number of multiplications and divisions, for any style of arithmetic, IEEE standard or not. In some circumstances, greater accuracy and speed can be achieved by using the formula

$$| x^i | = \exp(i \log(|x|))$$

whenever i is larger than some suitable threshold.
~~~~~~~~~~~~~~~~~~~