Why must $0^0 = 1$?

W. Kahan
July 27, 1988

Programs to compute x^y exist in the run-time support libraries of many a computer system, but those programs tend to disagree about what to do with 0^0 and even $(-3)^3$, sometimes producing reasonable values like 1 and -27 respectively, often unreasonable values like 0 and +27, usually warning messages, and perhaps a mixture of the three. The purpose of this note is to present a rationale for treating the special cases of x^y in a uniform way regardless of the programming language and the underlying computer hardware. The rationale has three components. One is historical, based upon a decent regard for past practice even when it has been illogical. Another component is algebraic, based upon the axioms that are most often found in texts; these axioms define x^j for integer variables j. The third component, analytic, defines x^y when y is not an integer.

Our goal should be *Intellectual Economy*, which translates here into consistency with a parsimonious set of axioms or principles from which anyone can easily deduce how to reach his goals without being obliged to memorize too much. The path to our simple goal is beset by complications. Some are posed by *singularities* like 0, which violates a few of the laws by which all other numbers abide, and ∞, which is ambiguous too. More onerous for the users of the run-time library, and a major challenge to its implementor, are complications posed by the finite limits that circumscribe all computers. These limitations are associated with *roundoff, over/underflow* thresholds and, in many computers, no way to express ∞ nor the concept called *NaN* (*N*ot-*a*-*N*umber) by the IEEE standards 754 and 854, called *Indefinite* by CRAYs and CDC Cybers, and called *Reserved Operand* on DEC PDP-11s and VAXs. These limitations would be less bothersome if they did not vary so much from one computer family to another; but because they do vary, we shall pursue an exculpatory policy, absolving the implementor of blame if his version of x^y is scarcely worse than it has to be in consequence of his computer's limitations.

Notation:

In keeping with the Fortran tradition, the letters i, j, k, l, m and n will be reserved for integer valued variables; we shall further reserve m and n for positive integers.

Let R denote the finite real numbers and R' the nonzero finite reals. Let the symbols "$\pm\infty$" represent $\pm$*Infinity*; we'll omit the signs just when they do not matter. Next let X denote that extension of the real numbers that includes Infinity; that is

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

The letters a, b, c, ..., g, h are reserved for finite real (R) and the letters p, q, r, ..., x, y, z for extended real (X) variables. These reservations will be altered later when complex numbers appear. Finally, the symbol "NaN" stands for "Not-a-Number" or "Indefinite" or "Reserved Operand", and is the only thing in our universe that can violate the identity $x = x$.

Below are the Multiplication and Division tables for a system $X \cup$ NaN consisting of the extended reals and NaNs. (No occasion will arise herein to add or subtract these entities.) The tables' columns have been so ordered as to render obvious the symmetry implied by the identity $x/y = x \cdot (1/y)$. In the tables, certain side-effects are indicated by asterisks *. Instances marked NaN * also signal "Invalid Operation"; the instance marked ∞ * also signals "Divide-by-Zero", which actually signifies "Infinity created precisely from finite operands". The signals are side-effects; other signals may be associated with overflow, underflow and/or *Inexact Operation* on some computers, as we shall now digress to explain.

MULTIPLICATION $x \cdot y$

	$y = 0$	y in R'	$y = \infty$	y is NaN
$x = 0$	0		NaN *	
x in R'		$x \cdot y$		y
$x = \infty$	NaN *		∞	
x is NaN		x		x or y

DIVISION x/y

	$y = \infty$	y in R'	$y = 0$	y is NaN
$x = 0$	0		NaN *	
x in R'		x/y	∞ *	y
$x = \infty$	NaN *	∞		
x is NaN		x		x or y

Exceptions, Defaults, Signals, and other Side-Effects:
These topics deserve a comprehensive and definitive treatment, but no such thing is available yet. What follows instead is an attempt to acquaint the reader with the issues and to bring some order to an incoherent scene. Brevity has not been achieved.

The laws of arithmetic cannot always be obeyed punctiliously. Exceptions must arise either because of singularities like 0/0 or because of limitations like roundoff and over/underflow. When an exception occurs (we say it is *signaled*), certain concomitant events may occur too, depending upon the provenance of the computer and its programming language:

A value may be delivered. For example, machines that conform to the IEEE standards get ∞ for 1/0 and NaN for 0/0; but IBM 370 VS Fortran specifies the overflow threshold 7.2×10^{75} for 1.0/0.0 and 0.0 for 0.0/0.0, while APL specifies $0/0 = 1$. Many otherwise respectable machines underflow quietly to 0. All these values, delivered in the absence of explicit requests for something else, are called *Default* values; the ones marked by asterisks in the tables above are specified by IEEE standards. Some machines lack any NaN and/or ∞ (IBM 370s lack both), so they have to deliver some other default values — for instance, a huge value in lieu of ∞. Some computing environments permit a programmer or user to substitute his own choice in lieu of any default value; we call doing so *presubstitution* whenever it must precede the exception. (Presubstitution is implementable, with suitable but inexpensive hardware support, on machines that are obliged by pipelining and other forms of concurrency to interrupt *imprecisely* on an exception, leaving its locus vague. Other

kinds of substitution require strictly sequential execution, which is slower or more expensive or both.)

A flag may be raised. As a side-effect, the exception may alter a variable that the program can later test to discover that that exception occurred. *ERRNO* in the language C is an example of a flag. The IEEE standards specify five flags, one for each of the five exceptions INVALID-OPERATION, OVERFLOW, DIVIDE-BY-ZERO, UNDERFLOW and INEXACT, but the specifications are vague about the types of these variables, which may be integers or pointers rather than merely boolean. Most computing environments have no flags accessible by software from higher-level languages; some computers lack the hardware to support certain flags at all. For example, none but the IEEE standards' machines can support INEXACT flags. Underflow cannot be detected on CDC Cybers and CRAYs except by testing for 0 among default results; and the same is true for most practical purposes on IBM 370's and DEC VAXs because their compilers disable the underflow traps by default.

A trap may occur. This will deflect the path of the program's execution to a *trap-handler*, if one has been prescribed, or else terminate execution. Many computing environments provide no easy alternative to termination for INVALID-OPERATION, OVERFLOW and DIVIDE-BY-ZERO. Some programming languages offer statements that begin with something like

ON ERROR ... (BASIC) or ON OVERFLOW ... (PL-1)

to permit a limited degree of trap-handling for those exceptions that the hardware can detect. In ADA, some exceptions execute an implicit RAISE NUMERIC_ERROR statement, which is the way ADA programs trap to a trap-handler that has been coded after an

EXCEPTION WHEN NUMERIC_ERROR ⇒ ...

statement at the end of a block of code. Standardized language support for trapping must be undermined by uncertainty about two issues. First, where may the program go after the handler? An ADA program must exit the block. BASIC dialects let programs RESUME execution, some at, some after the statement or line that trapped, but the dialects disagree about the syntax and/or scope (over which statements does a handler reign?) of trap handling. Second, which events will precipitate traps? An ADA program run on a CDC Cyber or CRAY cannot RAISE NUMERIC_ERROR for underflow. An ADA programmer who finds a default ∞ or huge number for (nonzero)/0.0 quotients satisfactory can declare

PRAGMA SUPPRESS(DIVISION_CHECK),

but only at the risk of losing control over 0.0/0.0 quotients. That the two kinds of quotients deserve different treatment is acknowledged by authorities as diverse as the IEEE standards, APL, and IBM 370 VS Fortran.

By now the reader should be persuaded that the computing world is not of one mind about exceptions. Perhaps they are characterized by that diversity, so that the only thing we may say about them without fear of contradiction is this ...

> *Fundamental Principle of Exceptions:* To any simple policy, enunciated in advance to cope with a class of exceptions, there is always someone who has good reason to take exception.

If exception handling is to progress from psychology to science, a taxonomy that identifies kinds of exceptions and distinguishes them from their presumed causes must be established. Such a taxonomy is provided by the IEEE standards for floating-point arithmetic; as summarized below, it applies to non-conforming machines too. The five-letter names supplied below are, alas, not specified by the standards, but are offered here until better suggestions come forward. Each name identifies a *flag* that, if the machine has it, records when at least one exception of that name has occurred.

INXCT:
Inexact Result is signaled when a computed result is in error due to over/underflow or roundoff; only machines that conform to the IEEE standards are equipped with hardware to record INXCT. The error due to roundoff would normally be ignored; but not during integer computations in floating-point when integers grow so big that their last few bits are rounded off, and not during a few exotic computations involving something called "preconditioning." The error due to roundoff is not usually ascertained, but cannot exceed the *uncertainty* due to roundoff, which amounts to half the gap between the computed result and its nearest representable neighbors on machines that round *correctly.* Machines conforming to the IEEE standards do round the five algebraic operations $+$, $-$, $\cdot$, $/$ and $\surd$ correctly; many other machines do not, and their bigger uncertainties can also be much bigger for some of these operations than for others.

UNFLO:
Underflow is signaled during an attempt to compute a nonzero result that is too close to 0.0 to be computed in the normal way. If the default result for every underflow is 0.0, we say "underflow flushes to zero," and then the uncertainty introduced by underflow can grossly exceed the uncertainty due to roundoff in numbers slightly bigger than the underflow threshold; this is what happens on almost all machines that do not conform to the IEEE standards. Conforming machines underflow "gradually;" this means that the absolute uncertainty introduced by underflow never exceeds the absolute uncertainty due to roundoff. CRAYs and CDC Cybers lack the hardware to record UNFLO. The CDC Cyber 17x machines suffer from "partial underflow:" tiny numbers closer to 0.0 than twice the underflow threshold behave normally for adds, subtracts and compares, but are mistaken for 0.0 by divides and multiplies.

DIVBZ:
Divide-by-Zero is signaled whenever the result to be computed from finite operands should be ∞ exactly except possibly for the absence of that symbol from the machine's floating-point numbers; among those deficient machines the default result, if any, is typically one of the overflow thresholds. Therefore DIVBZ ought to be signaled not only for 3.0/0.0 but also for log(0.0) and arctanh(1.0). Some computers confuse DIVBZ with OVFLO and/or INVLD below.

OVFLO:
Overflow is signaled during an attempt to compute a finite result that is too big to be computed in the normal way and must instead be replaced by a default result, either $\pm\infty$ or an overflow threshold, if computation is to continue. The CRAY's overflow thresholds are ambiguous because multiply overflows at results of which some are unexceptional for add/subtract, and divide overflows if the divisor is too small too.

INVLD:
Invalid Operation is signaled when any other signal (or none) would be more likely to mislead than help; its default value, a NaN, Indefinite or Reserved Operand, is preferable to any other for a similar reason but unavailable on some machines, for which the choice of default should ideally be different from any other values that the operation could produce validly. Different kinds of Invalid Operations can be worth distinguishing even though they all raise the same INVLD flag:

ZOVRZ:	0.0/0.0
IOVRI:	∞/∞
INVDV:	Either of the two invalid divisions above.
ZTMSI:	$0.0\cdot\infty$
IMINI:	$\infty - \infty$
FODOM:	Function computed outside its domain; e.g. $\sqrt{(-3)}$.

The foregoing exceptions are *phenomena* that we distinguish from their *causes*. Every INVLD, OVFLO or DIVBZ exception can be attributed to a *singularity*, as we shall explain after we note the last side-effect here:

± 0 :
The Sign of Zero is a side-effect specified closely by the IEEE standards but ignored by other arithmetics. It can be ignored too by programmers who do not care about it, but matters crucially to others, especially those concerned with complex arithmetic and efficient representations of conformal maps of slitted domains; more about that under *Topological Removal of a Singularity* below. For now, care taken with the sign of zero will be justified by asking whether identities like $b/(b/x) \approx x$ can be satisfied at least roughly for all extended reals including $x = \pm\infty$? YES on a machine conforming to the IEEE standards, otherwise NO.

Singularities:
All normal floating-point computations are intended to approximate piecewise *analytic* functions. The domain of such a function is a (normally finite) collection of regions; throughout some open neighborhood of every point in the relative interior of a region, the function may be represented by a convergent power series. A *singularity* of the function is one of those regions' boundary points; there the function or a derivative is usually undefined. Computing the function *at* a singularity might well signal INVLD or DIVBZ; *near* a singularity, OVFLO might be encountered.

Some singularities are *removable* by (re)definition of either the function's value or the region's topology. Defining $\sin(x)/x$ to be 1 when $x = 0$, or $(x^3 - y^3)/(x - y)$ to be $3y^2$ when $x = y$, illustrates (re)definition of value. Redefinition of topology is more subtle: consider for example all the rational functions of a real variable x; their singularities are all *poles* where the functions and their derivatives become infinite. Now *close* the real axis by adjoining the single point at ∞; one way to do so is to identify x on the real axis with $\theta = 2\arctan(x)$ on the circle, identifying $x = \pm\infty$ with a single point $\theta = \pm\pi$ on the circle. In this closed topology, every rational function $f(x)$ is freed from singularities by being identified with an analytic map $2\arctan(f(\tan(\theta/2)))$ of the circle to itself, thus turning ∞ into a point as unexceptional as 0.

Most singularities cannot be removed. This is fortunate because a world without singularities is a world without Physics. Attempts to bypass singularities amount to attempts to extend a function's definition beyond its original domain in a *natural* way, a way that is not arbitrary but is predictable by a thoughtful person. No such way need exist. For example consider $\sqrt{x}$, which is well defined as a real function only for $x \geq 0$. To avoid signalling INVLD or FODOM when $x < 0$, many a well-intentioned computer hacker has extended the definition of $\sqrt{x}$; three examples are

$$\sqrt{x} := \sqrt{|x|}, \quad \text{or} \quad \text{sign}(x) \cdot \sqrt{|x|}, \quad \text{or} \quad 0.5 \cdot (1 + \text{sign}(x)) \cdot \sqrt{|x|}.$$

The third (from an old UNIX math. library) is abominable, but it is merely the average of the previous two, and who can say which of those is better? None of the three satisfies the crucial relationship $(\sqrt{x})^2 = x$ when $x < 0$; that can be satisfied only by an extension to complex variables. Introducing those unbidden into a world of real computation is no panacea either, because the complex extension of $\sqrt{x}$ must violate some familiar identity like $(\sqrt{x}) \cdot (\sqrt{y}) = \sqrt{(x \cdot y)}$ which, because it is always valid for nonnegative arguments x and y, might mistakenly be presumed valid for the extension too. Such a mistake can give rise to an obscure anomaly like the following:

> Let $f(x) := \sqrt{(x + 1)}$; $g(x) := \sqrt{(x - 1)}$; $h(x) := \sqrt{(x^2 - 1)}$; in the absence of tests or absolute value signs, we might take the identity $f(x) \cdot g(x) = h(x)$ for granted since it is valid (to within roundoff) for all $x \geq 1$. But when $x \leq -1$ then $f(x) \cdot g(x)$ is real although $f(x) \cdot g(x) = -h(x)$.

That is why the least confusing choice for $\sqrt{(-4)}$ is NaN and an INVLD or FODOM signal. In general, the removal of singularities by extension of definitions is, like cosmetic surgery, to be undertaken only with good reason and then with good taste.

The mathematical procedures that remove singularities have their computational analogs. Programmers can (re)define functions in two ways. The most familiar ways entail tests and branches or conditional assignments like

$$f(x) := \text{ if } x = 0 \text{ then } 3 \text{ else } \sin(3 \cdot x)/x;$$

a better way entails *presubstitution* to avoid explicit tests:

> Presubstitute 3 for ZOVRZ;
> ... $\sin(3 \cdot x)/x$....

(An early but unwise instance of presubstitution is APL's rule that $0/0 = 1$, but the foregoing instance shows why that can be dangerous as a universal policy.) Presubstitution is preferable to explicit tests whenever the presubstituted values change only rarely, since it minimizes the time wasted upon operations whose unexceptional status would otherwise have to be confirmed by test.

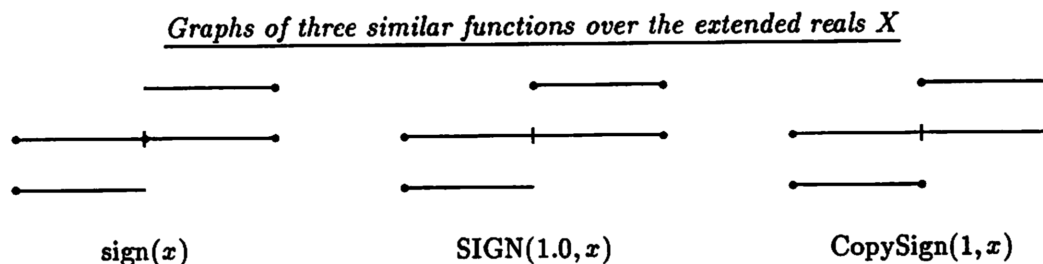
Topological Removal of a Singularity entails two concepts that go beyond merely closing a function's domain by incorporating into it its boundary points. One concept entails a choice of *metric*; the second concerns *ambiguity*. The criterion by which we decide when two numbers are so close that they may be regarded as practically indistinguishable reflects

our choice of metric. To ignore UNFLO is to choose a metric by which all sufficiently tiny numbers are regarded as practically indistinguishable from 0.0. A change of metric occurs when we go from the finite reals R to the extended reals X in so far as we then regard all sufficiently big numbers as neighbors (close approximations) of ∞, and then we may ignore OVFLO .

Pernicious ambiguity arises when we close the real axis with just one point at ∞. Programmers must take care that $+\infty$ and $-\infty$ are then treated as *equal*, which can be a nuisance to remember, somewhat relieved by a function PROJ(x) that maps every infinite x onto $+\infty$ and leaves every other x unchanged.

Benign ambiguity can arise when we close the graph of a function with a discontinuity. (The reason for closing a graph is to put all its extreme points onto it.) For example, the graph of the expression $x/|x|$ consists of two open pieces coinciding with the graph of sign(x) for finite nonzero x in R', an open set. The graph of sign(x) is closed at its distal ends where $x = \pm\infty$ but not at $x = 0$, where sign(x) is discontinuous; assigning sign(0) = 0, as is customary, defines sign(x) over a closed set X, the extended reals, but leaves the graph of sign(x) in three pieces, one dot and two half-open line segments:

Graphs of three similar functions over the extended reals X



The Fortran intrinsic function SIGN(1.0, x) matches sign(x) except for SIGN(1.0, 0.0) = 1.0; its graph has two pieces, one a closed line segment and the other half-open. The IEEE standard function CopySign(1, x) matches the others except at ± 0 where CopySign(1,+0) = +1 and CopySign(1,–0) = –0; this ambiguity gives it a closed graph on a domain where $x = 0$ is represented twice. Thus, when they are available, CopySign and ± 0 offer a programmer a convenient way to distinguish the sides of a boundary separating two line segments, areas or volumes without having to carry explicitly an auxiliary variable that is significant only on the boundary. In fact, the IEEE standards' specifications for zero's sign work so automatically as often to expunge an obscure bug that infests programs, written originally for other machines, to handle conformal maps of slitted domains in the complex plane. The bug causes parts of a mapped region's boundary to go astray, but not on an IEEE machine; see my paper *Branch Cuts for Complex Elementary Functions.*

Topological redefinition affects programmers in another subtle way that transcends presubstitution or the acceptance of defaults; it influences the programmer's attitude to the flags that memorialize the occurence of certain exceptions. For instance, having closed the real axis at ∞, a programmer will ignore DIVBZ but not INVLD. (This can be difficult on certain machines, CRAYs and INMOS T800 Transputers among them, that raise the same flag for both exceptions.)

More generally, at least for programs intended to run in IEEE standard environments, one can almost determine what domain the programmer had in mind by finding out which exceptions his program ignores. A program that ignores none of them, not even INXCT, is probably using floating-point arithmetic for integers wider than can be handled conveniently by the environment's fixed-point variables. (The *COMP* format on APPLE computers is a 64-bit integer format implemented in floating-point hardware when that is fastest, in which case INXCT gets translated into *Integer Overflow.*) Most programs ignore the INXCT exception when they are approximating real numbers by floating-point numbers. They ignore UNFLO too when the programmer believes that all errors with sufficiently small *absolute* values are acceptable; but if errors must stay *relatively* small then neither UNFLO nor OVFLO can be ignored. DIVBZ can be ignored by computations that treat the real axis as closed at infinity, either by one point at ∞ as is appropriate for algebraic functions, or by two points $\pm\infty$ as is appropriate for transcendental functions like $\exp(x)$ that are discontinuous across ∞. Functions like $\arctan(x)$ that are one-sidedly continuous at $\pm\infty$ can accept arguments x that have overflowed to $\pm\infty$ and ignore OVFLO. But INVLD must never be ignored.

How does a program *not* ignore exceptions? By tests. In IEEE environments it can test for a default value before it is over-written, or test a flag somewhat later. That flag allows one program to defer judgment to a later program where the exception's consequences may be easier to assess, provided execution has not been terminated.

Programs intended to be portable to machines that lack support for flags have only three strategies available. They can test before every operation that could be exceptional and branch to avoid the exception; this strategy can entangle a program in spaghetti-like branches, but it keeps control within the program. Second, in some environments, the program can specify a trap-handler to try an alternative computation in lieu of one that has aborted; this strategy is really the same as the first except that the spaghetti is invisible and therefore harder to debug. Third, a program may test default values promptly to know when an exception has occurred. This is costly and risky. For instance, results contaminated by underflow must be detected by comparison against a tiny threshold chosen aptly for the computer, taking into account ambiguities associated with gradual, partial and operation-dependent underflows. Similar considerations apply to overflow with the added proviso that computation be not halted. Predicates like "(1.0E37)/(1.0+abs(x)) > 0.0", which is *false* when x is ∞ on any machine that has it but *true* for every finite x on all current computers, serve as portable ways to discriminate between infinite and finite values x. The simpler predicate "$x - x = x - x$" works on IEEE standard hardware where $\infty - \infty$ is NaN and NaN $\neq$ NaN, but other hardware may behave differently and, besides, a compiler too clever by half may simply discard that predicate. The predicate "$x = x$" could suffer a similar fate rather than discriminate between NaNs and numbers. In the light of the anarchic variation among the arithmetics of computers other than those that conform to the IEEE standards, language standards could regularize exception-handling better by adopting standard names for machine-dependent subsets of IEEE standard flags and defaults than by attempts to supplement the languages' control structures with invisible tests and branches.

Whatever the exceptions and side-effects that can arise during multiplication and division, and whatever the options that a computing environment provides for these exceptions, at

least these must apply equally well to exponentiation, although it will be subject to additional INVLD exceptions like $(-4.0)^{0.5}$ that cannot afflict multiplication and division. We do not have the power to specify too closely how everyone has to respond to exceptional exponentiations; instead we can at best specify which exponentials should *not* be exceptional, and why.

Exponentiation defined Algebraically:
Descartes' definition of 1637,

$$x^n = \underbrace{x \cdot x \cdot x \cdot \ldots \cdot x \cdot x \cdot x}_{n \text{ copies of } x},$$

is accepted universally only when n is a positive integer, and then x may be any extended real (or complex) number (or matrix). Our first objective is to extend Descarte's definition so that x^j will make sense for every integer j, positive or not. x^j should be an extended real whenever x is. Ideally the extension should be *parsimonious*, entailing as few assumptions as possible, and therefore implying a minimum of special cases. The definition that results from this parsimony is displayed in a table below, after which comes an explanation to justify those entries, in the middle column, that differ from other opinions.

EXPONENTIATION x^j

	$j < 0$	$j = 0$	$j > 0$
x in X	$(1/x)^{-j}$	1	x^j
x is NaN	x		x

Three aspects of the foregoing table deserve explanation. First is the assignment $x^0 = 1$ regardless of whether x is 0, ∞ or NaN. Second is the trouble caused by roundoff, which obliges us to use unobvious algorithms to achieve respectable accuracy in all cases. Third are the side-effects. One of them is the INXCT signal, which may have to be emitted undeservedly on occasions when inhibiting it is too expensive. Another is the DIVBZ signal, which must arise when $0^j = (1/0)^{-j} = \infty^{-j} = \infty$ is calculated for $j < 0$. And when x^j is finite and nonzero but too huge or too tiny to represent within the normal range of floating-point numbers, then OVFLO or UNFLO respectively must be signaled and a suitable default value must be supplied, with care not to signal OVFLO when UNFLO is deserved or vice-versa. Finally, the signs that the IEEE standards attach to ± 0 and $\pm\infty$ must be respected. All will be explained below.

A Parsimonious Set of Postulates:
Historically, the meaning of c^j when $j < 0$ appears to have evolved from the observation that $c^{m+n} = (c^m) \cdot (c^n)$ for all positive integers m and n, followed by an insistence that this equation be satisfied when m and n are replaced by arbitrary integers i and j regardless of their signs. This approach turns out to be incapable of defining 0^0, so it has led some people to infer illogically that 0^0 must be undefinable. Let us follow this approach for a while to see what happens.

We postulate first the existence of a function $P(x, i)$ whose values, when defined, lie in X for all x in X and every i in I, and satisfy the noncontroversial defining relations

1. $P(x,1) = x$, and
2. $P(x, i+j) = P(x,i) \cdot P(x,j)$ unless this product is undefined.

Relation 2 allows for the possibility that the left-hand side might be defined even when one of the factors or their product on the right is not. We hope to deduce that $P(x,j) = x^j$. To this end, relation 1 is needed to preclude that $P(x,j)$ be $x^{k \cdot j}$ for some arbitrarily fixed but perverse $k \neq 1$. From these two relations soon follow familiar inferences:

$$P(x,n) = x^n = \underbrace{x \cdot x \cdot x \cdot \ldots \cdot x \cdot x \cdot x}_{n \text{ times}} \text{ for every } n > 0.$$
$$P(x, i \cdot n) = P(P(x,i), n) = P(x,i)^n \text{ for every } n > 0.$$

For every integer i, positive or negative or zero, and for all nonzero finite b and c in R'

$$P(b \cdot c, i) = P(b,i) \cdot P(c,i) \text{ and}$$
$$P(c, -i) = 1/P(c,i) = P(1/c, i) \text{ and } P(c,0) = 1.$$

Therefore the two relations above do define $P(x,j) = x^j$ either when $j > 0$ or when x lies in R'; but the remaining cases

$$P(0,0),\ P(\infty,0), P(0,-n) \text{ and } P(\infty,-n) \text{ when } -n < 0$$

are left ambiguous. The clearly undesirable assignments

$$P(0,0) = P(0,-n) = 0 \text{ and } P(\infty,0) = P(\infty,-n) = \infty$$

are no less compatible with relations 1 and 2 then are the more reasonable assignments for respectively 0^0, 0^{-n}, ∞^0 and ∞^{-n} in the table above. This ambiguity is intolerable mostly because it is unnecessary; just because 0^j is not defined unambiguously for $j < 0$ by one set of postulates, however familiar, is no reason to deny some other familiar and equally parsimonious set an opportunity to do the job.

Here is a pair of postulates almost as familiar and equally as parsimonious as the previous pair:

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

And this pair *majorizes* the previous pair in the sense that both agree on the definition of $P(x,j) = x^j$ wherever the previous pair defined it, but this pair defines it for all extended real x, finite or not, and all integers j regardless of sign. The table above is drawn from these defining relations.

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). Neither are they the only set of postulates that define $P(x,j) = x^j$. However, every other set that defines $P(x,j)$ unambiguously at $x = 0$ and $x = \infty$ regardless of the sign of j, differing only at $P(0,0)$ and $P(\infty,0)$, must imply postulate I for all other x; in so far as it uses an additional postulate or predicate to

(un)define $P(0,0)$ and $P(\infty,0)$ different from 1, such a set cannot be so brief as postulates I and II.

Finally, postulate I is consistent with other practices common in texts and the mathematical literature. For one example, the polynomial $a_0 + a_1x + a_2x^2 + \ldots + a_nx^n$ is often abbreviated $\Sigma_0^n a_j x^j$, which is valid at $x = 0$ only if $0^0 = 1$. Another example is the conventional interpretation of sums and products

$$\sum_1^n s_j = s_1 + s_2 + \ldots + s_n \text{ and } \prod_1^n t_j = t_1 \cdot t_2 \cdot \ldots \cdot t_n$$

when they are empty (when $n = 0$); $\sum_1^0 s_m = 0$ and $\prod_1^0 t_m = 1$. (0 and 1 are the respective identities for + and · operators.) Therefore the identification of x^n with that product when every $t_m = x$, valid for $n \geq 0$, would be violated at $n = 0$ by any other definition of x^n inconsistent with postulate I. Thus, postulate I provides the unique acceptable definition 1 for 0^0 and ∞^0, if they are to be defined at all; therefore, leaving them undefined seems more wilful than prudent.

A proof that postulates I and II are effective over the extended reals X has not appeared elsewhere that I know, so here it is:

By induction, as in Sigler's text, we may prove for every j in I and all x in R' that $P(x,j) = x^j$. Also easy to prove is $P(0,n) = 0^n = 0$ and $P(\infty,n) = \infty^n = \infty$ for all integers $n > 0$. The remaining cases are $P(0,-n)$ and $P(\infty,-n)$. Now, $P(0,-1)$ cannot be finite lest $1 = P(0,1-1) = 0 \cdot P(0,-1) = 0$; therefore $P(0,-1) = \infty = 0^{-1}$. Similarly, $P(\infty,-1) = 0 = \infty^{-1}$. Then 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.

The answer is the same whether we think of ∞ as an integer or not. If ∞ is not an integer then the two expressions above are defined by (3) above to be respectively infinite and zero, and then $+\infty$ and $+0$ are the signed values that follow the rule of *Signs* two paragraphs ago. If ∞ is an integer, we must decide whether it is even or odd; since all sufficiently big floating-point numbers in North American computers are even integers, taking the limit implies the same for ∞. Therefore $(-3)^{+\infty} = +\infty$ and $(-3)^{-\infty} = +0$. This conclusion could change on ternary computers, but programmers are used to that kind of risk.

A program for Real x^y :
Here are the specifications to be met:

EXPONENTIATION x^y with y = integer j

	$j < 0$	$j = 0$	$j > 0$
x in X	$(1/x)^{-j}$	1	x^j
x is NaN	x		x

EXPONENTIATION x^y with $y \neq$ integer

	$y = -\infty$	$-\infty < y < 0$	$0 < y < +\infty$	$y = +\infty$
$x < -1$	+0	NaN *	NaN *	$+\infty$
$x = -1$	NaN *	NaN *	NaN *	NaN *
$-1 < x < 0$	$+\infty$	NaN *	NaN *	+0
$x = 0$	$+\infty$	$+\infty$ *	+0	+0
$0 < x < 1$	$+\infty$	$\exp(y \cdot \ln(x))$	$\exp(y \cdot \ln(x))$	+0
$x = 1$	NaN *	$\exp(y \cdot \ln(x))$	$\exp(y \cdot \ln(x))$	NaN *
$1 < x < +\infty$	+0	$\exp(y \cdot \ln(x))$	$\exp(y \cdot \ln(x))$	$+\infty$
$x = +\infty$	+0	+0	$+\infty$	$+\infty$
x is NaN	x	x	x	x

All entries in this table except $(x < 0)^{\pm\infty}$, $0^{(y>0)}$ and $0^{-\infty}$ are produced automatically, including the signals where marked by an *, by the expression $\exp(y \cdot \ln(x))$ provided it is evaluated in a way analogous to the specifications of the IEEE standards, and then the expression $\exp(\text{NaN} \cdot \ln(x))$ quietly produces NaN for x^{NaN} too. In the previous table 1/0 signals DIVBZ.

Fortran libraries often use two subroutines, one to compute $X**J$ by repeated squaring and multiplication for INTEGER J, another to compute $X**Y$ from an expression like EXP(Y·ALOG(X)) for REAL Y. We shall treat both cases together because speed and accuracy can be enhanced by using the first method when $|Y|$ is a small integer, and the second when $|J|$ is not. The first method loses roughly $|J|$ units in the last significant digit carried. The second is no worse than if it altered the last digit or two of Y; this implies that about as many significant digits will be lost in $X**Y$ as are needed to hold its exponent in its floating-point representation. This error can be substantially diminished at very low cost by carrying a few extra digits in a few operations; an example of a program that does so may be found in the 4.3 BSD UNIX Math. Library, and similar programs are in use by Apple, DEC, IBM and Sun in their Fortran libraries so the details will not be presented here. No such inexpensive way to diminish error is known for the first method that computes $X**J$ unless $|J|$ is kept very small. ($-1 \leq J \leq 2$ is easy.)

The program offered below is structured in a way that reflects the definitions above except that, in deference to Fortran, $P(x, j)$ is not defined recursively. Instead, a subfunction $Q(x, n)$ is defined to compute xn for small positive integers n using the squaring function x^2, which is assumed to be as atomic as $*$ (multiplication) so that the same program can be used later for complex variables and matrices. Then $Q(x, n)$ is invoked by $P(x, j)$ to compute x^j for integers j of arbitrary sign and small magnitude. Later the real functions ln(...) and exp(...) are assumed to be *atomic* in that they handle their exceptions just as if they were rational operations; the exceptions for $\ln(x)$ are INXCT when x is not 0, 1 or $+\infty$, DIVBZ when $x = 0$, and FODOM when $x < 0$; the exceptions for $\exp(x)$ are just INXCT when x is not 0, $+\infty$ or $-\infty$, and OVFLO and UNFLO.

One swapping function called Flag(xcept, oldflag) is used to sense, save and restore the exception flags; for example, OvSav := Flag *(*OVFLO, false *)* copies the OVFLO flag into the variable OvSav (of type flags – a combination of boolean and pointer), and then clears it to a null state. A similar trapping function called Ftrap is used to save the exception-handling modes, institute the defaults for exceptions, and then restore the exception-handling *status*

quo ante. The defaults are presumed to be those defined by the IEEE standards; other defaults, especially finite approximations to ∞, may entail rearrangement of the tests and branches. Even so, exceptions and other special cases are handled with scarcely more code than would appear in any efficient implementation of x^y regardless of the style of arithmetic, IEEE standard or not, and regardless of decisions about which cases should be exceptional.

All arguments to procedures are passed by value so that they may be overwritten. The scratch variables t and u can be extra-precise variables to yield better accuracy in the final result. All signals necessitated by exceptions occur at the sign ... * during the computation of t or u. These signals include INXCT, UNFLO and OVFLO for obvious reasons when necessary, and ...

DIVBZ	when $x = 0$ and $-\infty < y < 0$,
INVLD (ZTMSI)	when $\lvert x\rvert = 1$ and $\lvert y\rvert = \infty$,
or (FODOM)	when $x < 0$ and y is neither integer nor $\pm\infty$.

```
Real procedure x**y :
  Real value x, y;

  Real subprocedure Q(t,n):                    ... = t**n by repeated squaring,
    Real value t; integer value n;             ... only for n > 0.
    Real  u;
    while n is even do { t := t**2; n := n/2 }; ... *
    u := t;
    repeat { n := n/2 chopped to integer;
             if n=0 then return Q := u;        ... and exit.
             t := t**2;                        ...  *
             if n is odd then u := u * t }     ...  *
           until false ; ... i.e. forever.
     ...   n = 15 is the smallest n for which this program
     ...   performs more real multiplications than the fewest
     ...   necessary, namely 6 instead of 5. If it is to
     ...   be used only for very small n, a fast table-driven
     ...   version of this program should replace it, and then
     ...   another table of thresholds for |t| can be used to
     ...   forestall spurious overflow inside P(...,...).
     end  Q ;

  Real subprocedure P(x,j):                    ... x**j for integer j <> 0.
  Real value x; integer value j;
  Real t;
  Flags OvSav, UnSav;
  TrapHandlers OvTrpSav, UnTrpSav;             ... program pointers.
  If j < 0 then                                ... Stop spurious signals
         { OvSav := Flag(OVFLO,false);         ... when j < 0.;
           UnSav := Flag(UNFLO,false);
               OvTrpSav := Ftrap(OVFLO,default);
               UnTrpSav := Ftrap(UNFLO,default);
           t := Q(x,-j);
               OvTrpSav := Ftrap(OVFLO,OvTrpSav);
               UnTrpSav := Ftrap(OVFLO,UnTrpSav);
           If Flag(OVFLO,OvSav) or Flag(UNFLO,UnSav)
               then return P := Q(1/x,-j)
```

```
              else return P := 1/t }
          else return P := Q(x,j);
End P(x,i).

Real s, t;
If y=0 then return x**y := 1;                ... x**0 exit.
s := 1;                                      ... will record the sign of x**y.
If y is integer-valued then                  ... y must be finite ...
   { if |y| is 1 or 2 or not too big
       then return x**y := P(x,integer(y)); ... exit.
     if y is odd then s := CopySign(1,x);    ... = +/- 1.
     x := |x| }
   else if |y| = Inf or |x| = Inf then x := |x|;
If x = 0 then                                ... deal with exits that eschew ln(0) ...
   { if y = |y| then return x**y := s . 0;   ... 0  exit.
     if y = -Inf then return x**y := +Inf };... infinity exit.
t := exp(y * ln(x));                         ... computed with extra precision. ... *
Return x**y := s * t;                        ... last exit.
End x**y.                                    ... DANGER: NOT YET FULLY TESTED!
```

Acknowledgments:

The author's work has been supported in part, and from time to time, by the U. S. Office of Naval Research, the Air Force Office of Scientific Research, the Department of Energy, and the National Science Foundation. The greatest help has come from my colleagues and students, among them Prof. B. N. Parlett, Dr. J. T. Coonen, Prof. J. W. Demmel, Prof. R. J. Fateman, Dr. D. Hough, Z-S. Liu, S. McDonald, Dr. K-C. Ng and Dr. P-T. Tang.

Bibliography:

Apple Computer Inc. (1986) *Apple Numerics Manual*, Addison-Wesley, Reading, Mass.

W. J. Cody et al. (1984) "A Proposed Radix- and Word-length-independent Standard for Floating-point Arithmetic", *IEEE MICRO*, Aug. 1984, pp. 86-100.

R. Descartes (1637) *La Géométrie. ...* a^3 first appears.

L. Euler (1749) *De la controverse entre Mrs. Leibnitz et Bernoulli sur les logarithmes, négatifs et imaginaires.*

IBM VS Fortran Version 2 Release 3 (1988) *Programming Guide* ch. 5 and *Language and Library Reference* ch. 9, publications SC26-4221-3 and SC-4222-3 . (The second document specifies that (-2.0)**3.0 is exceptional with default value +8.0.)

IBM Elementary Math Library (1984) *Program Reference and Operations Manual*, publication SH20-2230-1. (This document specifies that (-2.0)**3.0 = –8.0 unexceptionally.)

W. Kahan (1987) "Branch Cuts for Complex Elementary Functions", ch. 7 in *The State of the Art of Numerical Analysis* ed. by A. Iserles and M. J. D. Powell, Oxford Univ. Press.

The Regents of the University of California (1985) 4.3 BSD Berkeley UNIX Math. Library.

L. E. Sigler (1976) *Algebra*, Springer-Verlag, New York.