Brian Hayes

Barry Castle, The Hare and the Tortoise, *1987*

Paradox Lost

Zeno of Elea has a strange effect on me. When I first learned of his paradoxes, I understood about half of what I read; a second look illuminated half of what I had missed the first time; with still another reading I began to grasp half of the remainder. It has gone on like that ever since. But now, after much earnest study, I feel confident I shall soon understand all of it.

In my latest reading of the commentaries on Zeno I have been struggling to divine the meaning of a certain mathematical function called *zeno*(x), which accepts a number as its input and ought to return another number as its output. The output, or value, of *zeno*(x) is defined by two rules:

Rule 1. If x is zero, return zero.

Rule 2. If x is not zero, return the value of *zeno*(x/2).

The nature of the *zeno* function can be made a little clearer if you work through a couple of examples. The value of *zeno*(0) is obviously zero; rule 1 says so explicitly. The situation gets more interesting for other values of x. In the case of *zeno*(4), for instance, rule 1 does not apply, and so you must resort to rule 2; it instructs you to return the value of *zeno*(4/2), or in other words *zeno*(2). What is the value of *zeno*(2)? You can find out by consulting the same two rules. Again rule 1 does not apply, and rule 2 returns the new value of *zeno*(1). Thus you are sent back to the rules yet again, in a process that will continue until you learn the final answer.

The *zeno* function illustrates a technique known in mathematics and computer science as recursion. A recursive function takes an intriguing approach to problem-solving. Given a sufficiently simple question such as What is the value of *zeno*(0)? it answers directly. For any other question, rather than trying to puzzle out the entire problem in one go, it asks itself an easier version of the same question. If the problem is to find the value of *zeno*(n), the recursive definition of *zeno* reduces the problem to finding the value of *zeno*(n/2). In specifying that n be divided by 2, *zeno* is sure to be getting ever closer to the special value zero, where the entire sequence of questions should finally converge on a definite answer.

These days the hard mental exercise of analyzing a recursive function is no longer necessary. There is no need to think; you can just write up the function as a computer program and let the machine answer all the questions. What happens when *zeno*(x) is submitted to the silicon oracle? If you encode the algorithm in a particular programming language and hand it over to a particular computer,

what value is returned? Given an ideal language and an ideal computer, what value *should* be returned? Is the algorithm a properly framed and well-behaved one? If not, where does the fault lie, and how can it be fixed?

In addressing these questions I suggest adopting the following strategy: first let us endeavor to get halfway to the answers; then, having reached that goal, let us try to advance half of the remaining distance, and so on, until all uncertainties are resolved. As a start, I propose setting aside Zeno's perplexing function temporarily and considering a simpler problem, one that is roughly half as hard to figure out.

Consider another mathematical function, named *divide*(*m*,*n*), which is intended to define the operation of division for the natural numbers (that is, the nonnegative integers 0, 1, 2,...). Given two natural numbers *m* and *n*, *divide*(*m*,*n*) is supposed to return the quotient of *m* and *n*, the value of *m* divided by *n*. An algorithm for performing the division can be expressed in terms of two rules:

Rule 1. If *m* is less than *n*, return zero.

Rule 2. If *m* is not less than *n*, return the value of *divide*[(*m*−*n*),*n*]+1.

Here again the procedure is a recursive one, and it is noteworthy for sticking close to the most primitive and trustworthy concepts of arithmetic. Division is carried out by repeated subtraction: The quotient of *m* and *n* is computed by counting how many times *n* can be subtracted from *m* before the result becomes less than zero. If *m* is already less than *n*, no subtraction is done, and an answer of zero is returned. Otherwise the answer is one more than the result of invoking *divide* on the inputs (*m*−*n*) and *n*. For example, rule 2 indicates that the value of *divide*(5,2) is equal to *divide*(3,2)+1; when the same rule is reapplied, *divide*(3,2) evaluates to *divide*(1,2)+1; finally, rule 1 states that *divide*(1,2) has the value zero. When you work back through the results, the initial inputs 5 and 2 give the output 1+1+0, or 2.

As in the case of *zeno*, there are many questions one might ask about *divide*. If it were written up as a computer program, could it be run efficiently? How does it compare with other algorithms for division? How might it be extended to handle other kinds of numbers, such as the negative integers? These issues must wait their turn, however, because a more urgent question has come up: What happens if the divisor is zero? The question is a troubling one in this case, for it appears that the *divide* procedure has no protection whatever against the taboo operation of division by zero.

Computer programmers have developed a standard remedy for problems of this kind: They insert a "guard clause" into the procedure. The guard checks the value of the divisor *n* and refuses to attempt the division if *n* is zero. The idea can be expressed by adding a new rule to the divide function:

Rule 0. If *n* is zero, return some value signaling an error.

Now the question becomes: What is the value best suited for signaling an error? What sort of thing should *divide* return if it is asked to perform an impossible act?

The one crucial requirement is that an error value be chosen that cannot be mistaken for the legitimate result of a division. The choices available depend in part on what programming language the *divide* procedure is written in. For example, in Lisp—a flexible and rather freewheeling language—the value that customarily signals any out-of-the-ordinary condition is a special object called *nil*, or (). *Nil* cannot be the result of a valid division, and so it serves as an unambiguous error signal. All one need do is make sure that every program that invokes the *divide* routine knows to check for a *nil* "result."

Unfortunately, the *nil* tactic offers little help if the *divide* routine must be written in one of the more finicky programming languages. A case in point is the language called Pascal: therein, every function must be declared as returning a particular type of value such as an integer or an alphabetic character; once the declaration is made, the function is never allowed to return a value of any other type. A Pascal version of *divide* would be declared as an integer-valued function, but then it cannot be made to return a nonnumeric value in the event of an error. About the best one can do is choose −1 as the error signal, since the quotient of two natural numbers can never be negative. Even that choice is ruled out if the algorithm for *divide* is made to accommodate negative integers. Then every legal value of the function (that is, every value allowed by the earlier declaration of type) is a possible outcome of a division, and no values are left over to signal an error.

The problem that arises here is not peculiar to *divide*. Consider the two main procedures for handling a data structure called a pushdown stack (the metaphor can be envisioned as a stack of plates resting on a spring-loaded shelf of the kind often seen in cafeterias): *push* adds an item to the stack, and *pop* removes and returns whatever item is currently at the top of the stack. The *pop* routine is the main troublemaker: it must somehow signal an error if it is asked to pop an empty stack. Suppose the programmer chooses the character string *empty* as the error signal. Then a *pop* operation that returns *empty* has two interpretations: perhaps there was an error, or perhaps the top location in the stack held the string *empty*. The latter interpretation may be unlikely—and by choosing a sufficiently weird error value one can make it even more unlikely—but computers, churning along at a million operations a second, have a way of discovering the unlikely.

Another example comes from computer operating systems that reserve a certain character code as an end-of-file marker. Any program that reads the file will know to stop when it comes to the marker. But no matter what character is chosen, sooner or later someone will want to create a file that includes it. (The first such file may well be the one holding the program that checks for the presence of the end-of-file marker.) Similarly, in programming languages that mark the end of a character string with a special code, it is awkward to deal with a string that has that code embedded somewhere in the middle.

Even having a value set aside for such special uses, as with *nil* in Lisp, is no panacea. Most dialects of Lisp provide a feature called a property list, in which the programmer can make note of various attributes of an object. For example, an object representing a computer might have properties such as speed, memory capacity and disk size. But suppose you inquire about the disk size of a certain computer, and the answer comes back *nil*? Does that mean disk size has not been defined for this computer, or does it mean the property is defined but it has the value *nil*? There is no direct way of telling, and that ambiguity is a classic source of bugs in Lisp programs.

Problems of the same kind occasionally turn up outside computer science. For some years I lived in a midwestern town that tested its tornado-warning sirens at 1:00 P.M. on the first Wednesday of every month. Whenever I heard the sirens wail, I would look at my watch and check the calendar—and then I would wonder how the town would alert the people if a tornado were spotted at 1:00 P.M. on the first Wednesday of the month.

Every one of these problems *can* be solved, given enough effort and ingenuity. But the usual solution to the end-of-file and end-of-string puzzles suggests just how convoluted and awkward the issues can become. Suppose the character "§" is chosen as the end-of-file marker and you want to create a file that incorporates "§" as an ordinary character. One answer is to designate another character, say "\", as an escape code, signaling that the next character in the file is not to be assigned any special meaning but treated instead as a normal character. Thus the two-character sequence "\§" is to be recognized as a plain "§". Giving "\" a special meaning, however, creates another problem: How can you include a back-

slash in a file as an ordinary character rather than as an escape code? The usual strategy is to decree that "\\" is to be taken as a single, normal backslash. Such a scheme might seem entirely too baroque to be taken seriously by practical people, but in fact millions of computers rely on a system just like it.

The problem of selecting an unambiguous error value has at least one general solution. The idea is to let the program that invokes an error-prone routine choose the most suitable error value. The *divide* function, for instance, can be rewritten to accept an extra argument—namely, the value to be returned in the event of a divide-by-zero error. Thus *divide* takes the form *divide*(*m*,*n*,*e*), where the third argument *e* is the specified error value. Now the rules that decide the value of *divide*(*m*,*n*,*e*) can be given as follows:

Rule 0. If *n* is zero, return *e*.

Rule 1. If *m* is less than *n*, return zero.

Rule 2. If *m* is not less than *n*, return the value of *divide*[(*m*−*n*),*n*,*e*]+1.

The revision has the added benefit that there is no need to make every program that calls *divide* as a subroutine recognize a universal error signal. Each caller can define its own error-reporting signal.

Of course it is still possible for a caller to choose an inappropriate error signal, if only through perversity. If the expression *divide*(*x*,*y*,1729) returns the value 1,729, the result could mean either that *y* is zero or that *x*/*y* is 1,729. Selecting a nonnumeric error value such as *nil* would eliminate this source of ambiguity.

In Pascal and other languages with strict type checking, *nil* is not an option, and so a slightly different approach is needed. The best plan I have been able to devise entails some wasted motion. Again the function *divide* gets redefined to accept an error signal as a third argument, though now the signal must be an integer. When *divide* is called as a subroutine, the main program supplies an arbitrary integer as the error indicator, as in *divide*(*x*,*y*,137). If the value returned by the expression is anything other than 137, there is no need to worry about a divide-by-zero error. If the value is 137, on the other hand, a further step is needed to establish the meaning of the result: *divide* gets called again, with the same values for the first two arguments but with a different error signal. If, say, *divide*(*x*,*y*,11) still returns 137, you know that 137 is the quotient of *x* and *y*. If it returns 11, you know that *y* is zero. (If it returns anything else, you know you're in trouble.)

Finding a suitable value to serve as an error signal can be viewed as a problem in information theory. At issue is the bandwidth, or information-carrying capacity, of the communications channel through which a function returns values to its caller. For example, a Pascal function declared to return a boolean result—that is, a result whose only possible values are *true* and *false*—communicates over a narrow channel; with just two possible return values, the channel capacity is exactly one bit. An integer-valued function has a much broader channel, typically with a range of 2^{16} or 2^{32} possible values. Nevertheless, if all 2^{16} or 2^{32} integer values are legal results of the function, the channel is saturated; it is carrying as much information as it can. Only if some integers are not proper function results is there spare capacity for error signals.

Some of the error-reporting methods I mentioned earlier work by increasing the capacity of a saturated channel. Allowing a Lisp version of *divide* to return *nil* expands the range of possible function values and so increases the communications bandwidth. Confirming an error signal by making a second call to a Pascal procedure enlarges the channel in a different way; it is a kind of time multiplexing, squeezing more information through the channel by sending two signals in succession.

There are many other ways of increasing the carrying capacity of a channel. For example, in Lisp the *divide* procedure could be made to return a list instead of a number: the list would include both the result of the division and a boolean flag indicating whether or not the result is valid. The flag constitutes one additional bit of information that is being transferred from callee back to caller. In Pascal a data structure called a record could be used for the same purpose.

With these and other stratagems for overloading or circumventing communications channels, it might appear that the problem of finding a suitable error value is essentially solved. The only issues left to be settled would seem to be those of syntactic form: how best to express an error-handling strategy in the program text. But I am not convinced that all the fundamental difficulties have been resolved, at least for languages such as Lisp that have no strong type checking. There is one circumstance in which expanding the capacity of the channel cannot work. Consider a function whose range of legal returned values includes everything that could conceivably be expressed in the language. Since the communications channel already has the maximum possible bandwidth, there is no way of opening it further to accommodate error signals.

A function that can return anything at all? Is it possible to write such an omnipotent monster? As a matter of fact, it is easy. A simple example is the identity function, which accepts a single argument and returns that argument unchanged. Such a function might be written *identity*(*x*), and the single rule determining its value would be

Rule 0. Return *x*.

In Lisp, *x* can stand for almost anything—a number, a list of numbers, a boolean value, a string of characters—and in every case *identity* will return the value of *x*. There is nothing that can be written in Lisp that cannot appear among the outputs. One might argue that here the error-signaling problem simply does not arise: because every conceivable object is a legitimate value of the function, there can never be any need for an error signal. But then what value should be returned by the expression *identity*[*divide*(1,0)]?

Even apart from all the entanglements of interprocedural communications, there is something troubling about *divide*'s use of specially distinguished returned values to signal an error. Consider the transaction from the point of view of the program that invokes *divide*. You ask for the quotient of 1 divided by 0, and the answer comes back *nil* or −1 or 1,729. But those values are all patent falsehoods; 1/0 is not equal to any of them.

At this point it might be helpful to take another look at the original, unadorned definition of *divide*(*m*,*n*):

Rule 1. If *m* is less than *n*, return zero.

Rule 2. If *m* is not less than *n*, return the value of *divide*[(*m*−*n*),*n*]+1.

I have been suggesting extraordinary measures for protecting that algorithm from the error of division by zero, and yet in an important sense the algorithm is already immune to error. It has the pleasant property that it will never give a wrong answer! If $m \geq 0$ and $n > 0$, *divide* returns the quotient of *m* and *n*. On the other hand, if $m \geq 0$ and $n = 0$, *divide* never returns. It goes on forever subtracting zero from *m*. In other words, there is no need to tell the procedure that division by zero is undefined; it is the procedure itself that tells you. The properties of division can be inferred from the action of the procedure rather than being imposed on it from the outside.

Thus it could well be argued that the entire effort to "fix" *divide* is misguided, on the grounds that the procedure was never broken in the first place. According to that view, what began as a simple and lucid procedure, which cannot give a wrong answer, has been encumbered with all manner of error-handling machinery. The outcome is a series of "corrected" versions, many of which give results that are at best subject to misinterpretation. The final state of the procedure hardly seems an improvement over the original one.

What is the practical consequence of such a view? Should all computer pro-

grams ignore error conditions, trusting them to work themselves out? Well, not in any software I'm going to use, please. (There is a big difference—an infinite difference—between "never a wrong answer" and "always a right answer.") On the other hand, it would be dull-witted to dismiss all nonterminating programs as mere blunders, unworthy of serious interest. The question of whether or not a program terminates is the deepest question in computer science. Indeed, if there were no nonterminating computations, and thus no problem of determining whether a given program would halt or not, computing would be a humdrum affair, rather like mathematics in a universe with a finite number of primes. It is the endless loops and the bottomless recursions that supply the indispensable glimpse of eternity.

Such considerations bring us back to Zeno's procedure, *zeno*(*x*). The recursive structure of the procedure, which at first looked sound enough, now appears suspect. On successive invocations the value of *x* gets progressively closer to zero, the terminating case; except for a recursion to come to an end, it not only must approach the terminating condition but must actually get there. Zeno's method of dichotomy—dividing *x* in half, then in half again, and so on—shows no prospect of ever reaching its goal.

It seems to be the odd fate of Zeno's propositions, however, that although they resist the subtlest arguments, they yield to the bluntest facts. Philosophers and physicists have kept up a debate on the paradoxes for 2,500 years, but in the meantime Achilles long ago passed the tortoise. Here, too, rude experience intervenes. If the *zeno* algorithm is written in a real programming language, and the program is run on a real computer, it turns out the program does terminate after all.

The outcome depends on how numbers are represented in the machine and on the semantics of the "/" operation. If *x* is confined to the natural numbers and if "/" is a synonym for the procedure defined here as *divide*, *zeno*(1) comes to a halt after just one full cycle, since *divide*(1,2) is equal to zero. Even *zeno*(1,1024) terminates after just eleven stages of recursion.

But surely the use of natural numbers violates the spirit of the puzzle. Zeno would urge that the problem be solved with values on the real number line. In digital computations real-numbered values are commonly approximated by floating-point numbers. If *x* is represented in floating-point notation, it takes on the successive values 1.0, 0.5, 0.25, 0.125,.... In the ideal world of mathematics these numbers certainly form a nonterminating series. In the world of computers, however, the dichotomy procedure stops and reports a value of zero after at most a few thousand iterations. The reason is that only a finite number of bits are allotted to the representation of floating-point numbers, and so sufficiently small magnitudes cannot be distinguished from zero.

There is something perverse about these results. With *divide* it was hard to guarantee that the program will always terminate, as one might like it to do. With *zeno* it was impossible to make the program run on indefinitely, as mathematics says it ought to do. Which of the programs, then, is the more seriously flawed?

Even when Zeno's ancient sophistries have been put to rest, there is more to be done with paradox and the computer. Consider a procedure even simpler in structure than *zeno*(*x*) or *divide*(*m*,*n*). The procedure, called *russell*(*p*), has only one rule, no looping constructs and no overt signs of recursion. It would appear to have a straight-through path of execution, with a rule that is evaluated only once. The rule for *russell*(*p*) is:

Rule 1. Return *not*(*p*(*p*)).

Russell is a special kind of procedure called a predicate: a procedure whose result is always a boolean value, either *true* or *false*. The argument of *russell* is also assumed to be a predicate. According to the definition, *russell*(*p*) returns the value *true* if and only if *p* is a predicate that has the property that it is not true of itself. For example, suppose there is a predicate called *number*, which yields *true* whenever it is applied to a number but yields *false* when given any other kind of object. Then *russell*(*number*) will return a value of *true*, because the procedure *number* is not itself a number. In contrast, one might have a predicate named *procedure*, which returns *true* when it is applied to any procedure but yields *false* otherwise; the expression *russell*(*procedure*) is *false*, because the predicate *procedure* is a procedure.

The question is: What value is returned when *russell* is invoked on itself, as in *russell*(*russell*)? Logic says the answer is *true* only if the answer is *false*. What should a computer say when it is asked such a question? ●

Brian Hayes is the editor of American Scientist. *He acknowledges the work of Joseph E. Stoy, R. D. Tennent, Philip Wadler, Robert S. Boyer and J Strother Moore, Daniel P. Friedman and David S. Wise, Robin Milner, C. S. Wetherill, Barbara Liskov, and Shaula Yemini and Daniel M. Berry.*

Steven Campbell, This Is the Final Warning These Sentences Are Not the Same, *1988*