Handling Arithmetic Exceptions

W. Kahan
Elect. Eng. & Computer Science,
University of California
Berkeley CA 94720

Abstract:

An *Exception* arises when an operation performed by a computer has to produce a result to which some people might reasonably take exception. Examples are *Division by Zero*, *Overflow* and Floating-Point *Underflow*. Though most (but not all) exceptions must be rare, too rare to be topics of everyday conversation, they are not so rare that computer programmers and users can ignore them altogether. This paper presents proposals, many of them now implemented on a few computers, to handle arithmetic exceptions in a generally satisfactory way at a tolerable cost. The proposals are designed to be fully compatible with concurrent, overlapped, parallel, pipelined and vectorized computing on new hardware that will be designed to support them without *precise interrupts*. *Flags* and *Modes* are proposed to help programmers cope with exceptions; *Retrospective Diagnostics* are proposed to help most of the rest of us, who aspire to use computers without having to program them. The features of IEEE Standards 754 and 854 are supported by but not obligatory for the proposals.

Introduction to Exceptions, their Defaults, and their Flags:

Words like *Error*, *Exception*, *Overflow*, *Invalid* and others to be mentioned later can refer to a single incident, a class of such incidents, or perhaps a state of mind. Equating *Exception* to *Error* is a mistake; an Exception becomes an Error only when it is handled badly. Some programming languages seem to have taken bad handling for granted, but they can be taught better; or else we can work around them without changing their compilers. Ideally, exception handling would be as independent of language as the meanings of phrases like " cos(x) " but for three issues:

<u>Scope:</u> If a program contains a statement that specifies some kind of exception handling, over what part of the program does that statement hold sway?

<u>Pre-emption:</u> Some languages provide exception handling statements that derail program control as soon as a run-time Error is detected; but whether any particular Exception is an Error may have been removed from the programmer's discretion by the designers of the hardware or of the language.

<u>Efficiency:</u> By taking cognizance of exception handling processes, a compiler could improve the speed and reliability of emitted machine code.

These issues are important, but they will be discussed only after the nature of exceptions and their flags have been explained.

Table 1 : Arithmetic Exceptions

Name	Description of Exception	...	Exceptional Value
ALLXS	ALL eXceptionS listed below (for treatment *en masse*)		
OVFLO	Floating-point OVerFLOw	....	$\pm\infty$ or a huge number
DIVBZ	Exact ∞ from finite operands, like 1/0	...	$\pm\infty$ or a huge number
UNFLO	Floating-point UNderFLOw	....	*Gradual underflow*, or 0
INXCT	INeXaCT	....	floating-point result rounded
INTXR	INTeger eXception or eRror like overflow or 1/0 with dubious result ?		
INVLD	INVaLiD operation, perhaps one listed below	....	NaN or ?
ZOVRZ	0.0/0.0	....	NaN or ?
IOVRI	∞ / ∞	....	NaN or ?
INVDV	One of the two INValid DiVisions above	....	NaN or ?
ZTMSI	$0.0 * \infty$	....	NaN or ?
IMINI	$\infty - \infty$	....	NaN or ?
FODOM	Function computed Outside its DOMain; e.g. $\sqrt{-3}$	....	NaN or ?
UNDTA	UNinitialized DaTum or vAriable	....	NaN or ?
DTSTR	Attempted access outside a DaTa STRucture	....	NaN or ?
NLPTR	De-referencing a NiL PoinTeR	....	NaN or ?

Table 1 exhibits a comprehensive (but not necessarily complete) list of exception classes, showing five-letter names for them. These names are so chosen in deference to a venerable programming language that is limited to six-letter names; our choices allow programmers the freedom to choose one more letter for the names of variables associated with the exceptions. Our names have five letters instead of fewer to lower the risk of collision with names already chosen for other purposes.

The default results suggested in Table 1 are consistent with those prescribed by the IEEE standards for five classes of exceptions, namely INVLD, OVFLO, DIVBZ, UNFLO and INXCT. The other exception classes are not mentioned explicitly by the standards. All the exception classes in Table 1 will be explained in more detail later, after we discuss *Flags*.

Each named exception can occur only when an expression or a variable cannot be given a numerical value without violating some rule that might reasonably have been expected to constrain that value. Whether rules devised by mortals deserve the same rigid obedience as Divine Law is a question for whose contemplation we make provision by suspending judgement rather than by terminating program execution; to this end, exceptional expressions and variables have to be assigned values with which the program can continue execution. On some machines those exceptional values are unpredictable except by someone who knows the hardware's wiring diagram; on other machines the designer's whimsy is documented for the benefit of programmers. Few conventions exist to keep exceptional values consistent from one machine to another, so the values provided in Table 1 are just suggestions.

Table 1 provides a glimpse at the real trouble with exception handling, - its diversity. The computing industry has spawned innumerable schemes to handle exceptions, each with its faithful adherents determined to follow their own *tao* rather than mine. We cannot cast their schemes upon the Scrap-heap of History without throwing out all their software too, and that waste is too high a price to pay for true enlightenment. Instead I hope, eclectically, to comprehend all the worthwhile exception handling schemes even if no single computer system's hardware is likely to support them all, even if programming languages and compilers all go their own ways, even if my hopes verge on the reconciliation of the irreconcilable. The next four paragraphs indicate how near to irreconcilable are existing schemes to-day, and foreshadow how I hope to reconcile them.

Exceptional Values:
The symbol "∞" appears in the last column of Table 1 to stand for a special floating-point number found on machines that conform to IEEE standards 754 and 854 as well as on CRAYs and CDC Cybers. Although ∞ can be simulated after a fashion on DEC VAX and PDP-11's, most other computers, IBM 370s among them, have no practical way to simulate ∞ , and must instead approximate it by the biggest floating-point number available. That is roughly 10^{76} on an IBM 370, not nearly so big as the bigger finite values found on many other machines, but acceptably big for many applications. Big finite numbers do not always behave like ∞ , nor do infinities on diverse machines that have them behave quite the same; for instance, $1/(1/(-\infty))$ might not yield $-\infty$ on machines that lack the -0 provided by the IEEE standards. But Table 1 ignores discrepancies like that.

The last column of Table 1 shows a question mark (?) or the symbol "NaN" for those exceptional values that cannot reasonably be approximated by $\pm\infty$ nor any finite value. The symbol "NaN" stands for "Not a Number," a special bit-pattern provided for certain exceptional floating-point numbers by IEEE standards 754 and 854. Analogous bit patterns are provided by some machines that do not conform to those standards; the CDC Cyber family and CRAYs have an "Indefinite" value; DEC VAX and PDP-11 machines have a "Reserved Operand." The analogies are imperfect; only the IEEE standards' NaNs behave in a fully predictable way in comparisons and some other contexts. Other computers, IBM 370s among them, have nothing comparable to NaN for floating-point variables; and very few computers have anything like NaN for integers. Therefore, the exceptional value must be regarded as undefined or unpredictable (represented by "?") whenever no NaN nor other natural value is available for it.

To be most useful, exceptional values must be predictable; they must be supplied *by Default*, which means that those predictable values ought always to be supplied except when a program requests something else explicitly. For instance, a machine that does possess infinities ought to supply one of them when 1.0/0.0 is computed by any program that contains no mention of DIVBZ (i. e. floating-point division of a nonzero by zero). A machine that possesses no infinities but can continue execution after DIVBZ might supply its biggest floating-point magnitude with an apt sign whenever it does so continue.

To Stop or Not to Stop:
Must 1/0 stop the machine? No. The default response to DIVBZ ought to be continued execution on machines that do have $\pm\infty$, but aborted execution otherwise. If not, if the default response to DIVBZ were always to terminate execution, a programmer who wished to distribute software that used ∞ would have to know the diverse magic words that enable its use on diverse machines. On the other hand, if the default response to division by zero were always to continue execution even if only with a huge finite value then quotients like $(1/x)/(4/x)$ at $x = 0$ could yield anomalous finite values without warning. The choice of a default response (continue, or abort) that correlates with the availability of a suitable default value (∞, or merely *huge*) is not so much a matter of taste as a vote of confidence (or otherwise) in the ability of the rest of the exception handling system to prevent calamities. When a program designed to exploit ∞ is run on a machine that balks at DIVBZ for lack of an infinity symbol, the program will presumably stop rather than deliver a final result that is invalidated by that lack. On the other hand, when a program expected to abort rather than continue after DIVBZ, but con⁺aining no explicit request to abort, is run on a machine that supplies ∞ and then continues to an invalid conclusion, then the *Retrospective Diagnostics* (to be explained later) should alert the user to "Unrequited Division by Zero" in his program.

The very idea that a program continue execution after division by zero must make some readers uneasy, while others will accept it without fuss. Similar differences of opinion are aroused by the other exceptions listed in Table 1 ; even their names are open to dispute. (If anyone knows better names, I shall receive their suggestions gratefully.) Some of the exceptions are undetectable on some machines; only those that conform to IEEE standards 754 ans 854 can detect INXCT ; and some machines, notably CRAYs and CDC Cybers, lack hardware to detect UNFLO . To cope with these unconformities we shall propose a *Menu System* that lets computers omit certain exception-handling capabilities from their menus but does not undermine the overall utility of our proposals. Just as one program works better on machines with wider range or precision or more memory, another program will work better on machines that are more sensitive to exceptions or more tolerant of them, but it will not malfunction misleadingly otherwise.

A few exceptions, notably UNDTA, DTSTR and NLPTR, require for their detection that compilers emit extra code to perform bounds checking or comparable operations at run time; these operations are obligatory in some strongly typed languages, but optional in others. We shall advocate the use of *Modes* to enable or disable the detection of various exceptions, implemented in some cases by compiler directives and in others by calls upon library programs at run-time. Other Modes will alter the exceptional values delivered by default; these Modes are made necessary by the impossibility of universal agreement upon those exceptional values (else they would not be regarded as exceptional).

Diverse opinions about exceptions extend also to strategies for handling them. The most primitive opinion would outlaw Errors/ Exceptions altogether, obliging a programmer to insert a test and branch into his program at every point where it was necessary to preclude them. In general, this strategy is so intolerably onerous that many programming languages provide an alternative among their control structures; they allow a program derailed by an Error to transfer control to an Error-Handler specified by the programmer beforehand. These control structures cannot, in general, identify the point of derailment precisely nor can they generally allow the program to resume execution at the point most convenient for the programmer. For instance, different dialects of the language BASIC derail at different places within a line or statement depending upon how an interpreter or a compiler may have rearranged the order of operations within that statement; consequently the ON ERROR GOSUB ... and RESUME statements often behave differently on different machines. On machines that execute operations concurrently in multiple arithmetic units or in a pipeline, the cost of a *Precise Interrupt* that would derail and resume precisely is the inhibition of concurrency to an extent that must intolerably penalize speed or the simplicity of the hardware or both. We need other strategies.

Another strategy for exception handling is implicit in exceptional values provided by default for exceptions that do not derail computation. Instead of testing and branching to preclude them, a program can test for the consequences of exceptions afterward. An ideal exceptional value would be so apposite as to require no tests; in this respect ∞ is usually apposite to division of a nonzero quantity by zero. An exceptional value cannot be always apposite (it would not be exceptional); therefore programs must exist that have to test for exceptions after they occur. But the tests may have to be postponed until so long after the event that no exceptional value is visible despite the damage it has done. For instance, the evaluation of $Q := (A+B)/(C+D)$ may produce zero, a value not exceptional in its own right, when the correct value would have been 0.25 but for the accidental overflow of C+D to ∞ before it could be divided into a huge A+B . On a machine that overflowed to the biggest available finite magnitude instead of ∞ , the evaluation of Q might yield 0.5 , which is more misleading than zero. This is one situation that cries out for an *Overflow Flag*.

<u>Why we need Flags:</u>
Another situation that implies the necessity of *Flags* involves a family of utility subprograms that access and update a complicated data structure without any awareness of the use to which the data will be put. For all that the utilities know, an exception that arises during an update might affect only some part of the data structure destined not to be used; therefore, to abort updating whenever an exception occurs would be to over-react. On the other hand, the programs that invoke the utilities can be expected to know whether certain exceptions matter. OVFLO or UNFLO occurring during the updating process might be important to the invoking

program, in which case it would have to scan the data structure for exceptional values after updating it. That scanning process could be as complicated and expensive as the update, but far less rewarding when exceptions are very rare. A better strategy for the invoking program is to discover (by testing summary Flags) whether OVFLO or UNFLO occurred during an update, and only then scan the data structure for entries that may be repaired while the data is still fresh. That is why Flags are necessary.

In short, a Flag is a signal that the computer has had to do something disputable, and if the program does not respond to that signal and absorb it then the program's user will have to judge whether to ignore the signal or not.

Flags constitute a data-type that shares some of the properties of the LOGICAL or BOOLEAN data-type in Fortran or Pascal, the POINTER data-type in Pascal or C, and the *external integer* ERRNO in C or certain *system variables* in APL. A flag can be *raised* or *lowered*; and when lowered it has the value FALSE in BOOLEAN contexts, NIL or NULL in POINTER contexts. A raised flag is TRUE in BOOLEAN contexts, but awkward to interpret as a pointer at run-time because it points into a *Log* of *Retrospective Diagnostics* that depend too much upon details of the computer system's implementation and limitations; more about that later. A language purist might insist that a variable of type *Flags*, say flag0, be coerced explicitly before it is used in a BOOLEAN context; he might protest that the statement

```
                    If  flag0  then ...
```

is linguistically unsafe whereas something like

```
                  If  BOOL(flag0)  then ...
```

is unexceptionable. He is right, but I prefer the simpler way.

Flags can be copied and lowered by assignment statements like

```
    flag1 := flag2 ;    flag2 := FALSE  (or NIL,  or NULL) ;
```

and they can be combined and tested like BOOLEAN variables:

```
        if  ( flag1 and (x = 0.0) )  then ...  else ... .
```

Raising flags directly, as in statements like

```
        flag3 := TRUE ;    flag4 := flag1 and (x = 0.0) ;
```

is legal but abnormal because the resulting flags may be useful only in BOOLEAN contexts, their POINTER values having been corrupted.

The System Flags:

Among the flags are certain *System Flags* accessible only through the flag-valued function FFLAG(*excep*, ...) . Its first argument *excep* must be the name of an exception in Table 1 ; for example FFLAG(UNFLO, ...) accesses the UNFLO_flag if it exists. But if the computer's hardware is incapable of detecting underflow then "UNFLO" will be an undefined name detectable at compile-time; if the compiler overlooks this lapse then the expression FFLAG(UNFLO, ...) will cause a DTSTR exception at run-time and, if that does not derail execution, return FALSE by default, after which only FFLAG(DTSTR, ...) or FFLAG(INVLD, ...), and perhaps an entry in the *Log* of *Retrospective Diagnostics*, will remain to defend the program's user from possible jeopardy.

The value returned by FFLAG(*excep*, ...) is always the current value of the system's *excep*_flag that memorializes exceptions of class *excep* provided they are detectable at run-time; no other way exists to access that flag. It is *sticky* in the sense that a system flag gets raised as a side-effect of an appropriate exception and stays raised until lowered by an explicit invocation of FFLAG for the purpose. My syntax for that invocation is idiosyncratic enough to deserve a digression here.

If "flag0" is "TRUE" or "FALSE" or the name of a variable of type *Flags*, then the expression FFLAG(*excep*, flag0) first returns the current value of the system's *excep*_flag and then sets *excep*_flag := flag0 . Hence, FFLAG swaps a new system-flag value for the old. This behavior has been chosen because the most common statements involving FFLAG will entail first the copying of a system-flag and its lowering, then a block of code containing operations that could raise that flag as a side-effect of an exception, then the restoration of the system-flag to its copied value, followed by a test that tells whether the block of code in question encountered an exception that raised the flag. In this way, any exception that occurs inside the block can be corrected and entirely hidden from the user of that block. Two swaps require only two invocations of FFLAG instead of four separate statements, two to read and two to write into the system flag. However, occasions do arise when a system flag is to be read but not changed, and then I omit the second argument flag0 from the invocation thus; FFLAG(*excep*) returns *excep*_flag but does not change it. My syntax must tax implementers who labor in languages that disallow variable-length argument lists except for favored functions like PRINT or MAX . They may use a dummy flag value DUMMY in place of flag0 to achieve the same effect as omitting it, or they may find another syntax better than mine.

While we are on the subject of syntax, we might as well digress to another language issue raised by *Functional Programming* in *Applicative Languages*. In their purest forms, these languages deal solely with functions, with no notion of *variable* or *state*. Side effects are anathema to these languages, so our system flags have to be treated as appendages to the values taken by functions. Since these values can be complex structures with many components, appending another component called *flags* is no great conceptual burden; the flags component of a function composed from other functions is a kind of logical sum of their flags components that summarizes the exceptions that have affected the function's value. When a default exceptional value turns out to be apposite or when a conditional assignment supersedes an unwanted exceptional value, the corresponding flag should be lowered lest it raise a needless doubt about a function's value. The syntax for lowering and raising flags in applicative languages lies beyond my present proposal, which is aimed at conventional procedural languages. Besides, I do not expect applicative and functional programming to supplant procedural languages but rather to coexist with them. Then exception flags will be handled most efficiently in the parts of a program that are procedural, whereas *Presubstitution* (a way to change exceptional values in advance that will be described later) will handle most exceptions in the applicative part better than an auxiliary flags component would.

The Old Ways are Not the Best Ways:

The foregoing complicated restrictions surrounding flags may seem superfluous to C programmers who use the external (global) integer ERRNO to reveal exceptions, or to assembly language programmers who routinely manipulate flag bits in a computer's status register. There are reasons for that complexity, but they are complicated too. Start with ERRNO ; it receives error codes from certain C library functions (exp, acos, ...) when they cannot yield an unexceptionable value but they continue execution anyway. A typical error code is the constant ERANGE written into ERRNO when a result should overflow; exp(1000000000000.0) does that. Another error code is the constant EDOM written into ERRNO when an argument lies outside the function's conventionally accepted domain, an example is acos(3.7) .

What's wrong with ERRNO ? Even if it revealed exceptions in rational arithmetic operations (it doesn't) as well as in library functions, ERRNO would have to be tested immediately after every potentially exceptional operation of interest lest a subsequent exception obscure the situation by overwriting ERRNO . But this test would force each such operation to finish before the next could begin. Of course, extra hardware could be added to a computer to retract prematurely initiated operations whenever (rarely) necessary, but that kind of added hardware slows down all operations a little in the hope that it will prevent a few of them from being slowed down a lot. On the fastest computers, tests and branches that require prior operations to finish first tend to inhibit concurrency and put bubbles into pipelines; that is why we seek to move branches out of tight loops. Therefore flags must be sticky to reveal after a loop all the kinds of exceptions that occurred inside it. ERRNO cannot do that.

ERRNO has to be abandoned, along with all other schemes that demand precise interrupts or too many explicit tests and branches, not so much because fast machines *cannot* be built to support them (they can!) as because most of the fastest machines *will not* be built to support them efficiently. Schemes that run inefficiently on those fastest machines will be avoided by ambitious programmers of codes intended for widespread distribution. Those programmers could use flags because, if well implemented, flags subtract far less from speed than do conscientious tests of ERRNO . As side-effects of exceptional concurrent operations, flags can be raised out of order; they need not be raised in the same sequence as the operations that raised them appear in source or object code. This means that *raising* a flag need not synchronize, nor inhibit concurrency, nor put bubbles in pipes. *Reading* a flag must synchronize; but flags are designed to be tested rarely, outside inner loops, at natural synchronization points in programs, so the cost of testing can be spread over many potential exceptions instead of a few.

Why must references to system flags have the syntax of function calls instead of mere references to variables? Assignments like

flag1 := *excep*_flag ; *excep*_flag := FALSE ;

are quite acceptable provided the compiler knows that these are synchronizing operations that must wait for prior operations to finish, and that changes to a system flag can have further side-effects (which will be explained in a moment). But a compiler that knew nothing about exceptions might "optimize" a reference to a flag by moving it ahead of a possibly exceptional operation. Since only the most reckless optimizer would move a function call whose side-effects are unknown to the compiler, putting system flags into function calls enhances the likelihood that they will function as intended regardless of whether the compiler cooperates with exception handling. This is important to anyone who would retrofit my kind of exception handling into an environment with pre-existing compilers that are best not tampered with.

Why are flags not merely BOOLEAN variables, nor simply single bits in a processor's status word? A flag has to be a pointer in order to provide the Retrospective Diagnostics mentioned above and to be described in detail later. Without them, continuing execution after an exception could induce dangerous consequences in programs imported from an environment where that exception always aborts execution. In my environment, a program that leaves a flag raised when it terminates is trying to say one of three things:

- o The flagged exception is ignorable because it was handled correctly by its default response; this case could be handled even more humanely if the programmer, knowing the flag to be ignorable, had lowered it at the end of his program.
- o The flagged exception is deserved by the program's results; for example, exp(1000000000000.0) should signal OVFLO on most machines. By testing FFLAG(OVFLO, ...) afterwards the program that called exp could decide whether OVFLO occurred and what to do about it. But this test might not happen; ...
- o The flagged exception was not anticipated by the program's user nor by its programmer, unless he expected it to abort execution. Whether the final results have been invalidated by that exception is now a question that will require some further investigation. That investigation begins with the flag, which points to an entry in the Log of Retrospective Diagnostics that, in turn, points to the site of the operation that was flagged exceptional, from which point debugging can begin.

Now the reason for treating a flag as a pointer should be clear. Whenever an exception raises its flag it must record this event and its location in the Log of Retrospective Diagnostics and set the flag to point to the Log entry, all of which takes a little time. It is not obvious that Logging is compatible with imprecise interrupts and concurrent execution, but it is true none the less, as will be explained later. What must be explained now is that a call to FFLAG(*excep*, flag0) has rather more to do than merely set *excep*_flag := flag0 , which is why a function call is needed instead of something that merely alters a bit in a status word.

If all exceptions were rare the time taken to Log them all would be inconsequential, but some exceptions are not that rare. INXCT occurs with every rounding error; and when one UNFLO or OVFLO occurs in a loop then many more are likely to follow. Fortunately only the first exception in a class need raise its flag and be Logged; while its flag stands raised all subsequent exceptions in that class may be ignored provided its exceptional operations do produce the exceptional value expected by default. Let us assume that the hardware is designed to deliver that desired exceptional value. Then raising a flag may also tell the hardware to stop sending further signals to raise that flag; lowering a flag must tell the hardware to send a signal when next that flag has to be raised. Lowering a flag, like reading it, is a synchronizing operation that must wait until all operations that could raise the flag have finished. Raising a flag does not synchronize. This remains so whether it is raised as a side effect of an exceptional operation or directly by a call upon FFLAG(..., TRUE) . Repeated signals to raise a flag will not affect its BOOLEAN value; and if the signals arrive out of order, as well they may when several operations are concurrent, the worst that can happen is that subsequent retrospective diagnosis will identify the first such exceptional operation to be detected instead of the first one issued.

Thus we see that the time spent raising flags need not much exceed the time a program spends lowering them. Therefore a program that pays no attention to its exceptions or commits none will spend very little time on them. Similarly, as we shall see later, the space occupied by the Log of Retrospective Diagnostics cannot much exceed the space occupied in a program by calls upon FFLAG , so the Log cannot consume too much memory either. And yet the user of a program oblivious to exceptions derives a measure of protection from their worst consequences because the flags left standing by the program, and the Retrospective Diagnostics to which they point, can tell him something about what has happened and where, should he later come to care.

Herein lies the principal value of my proposals. Most computer users are oblivious to exceptions until they occur. A user runs a program to get a result, and only when exceptions occur and deny him a result or undermine his confidence in it would he want to do anything about them. My proposals help him get what he wants. He need not be preoccupied about exceptions but may deal with them as afterthoughts, if they arise; and if their defaults have been chosen wisely, and if also most of the software he uses has been designed robustly to hide irrelevant exceptions, chances are good that what few exceptions come to his attention will be localized well enough that he can decide easily what to do about them.

Professional programmers have an obligation to protect their clients from unnecessary distractions like irrelevant or avoidable exceptions, but that obligation has in the past been so hard to discharge that we have had to forgive programmers when they failed to live up to our expectations. My proposals would make exception handling easier and more nearly portable, but still neither easy nor portable. Exception handling will never be an easy task. We can make it portable only by implementing as uniformly and as widely as possible those features necessary to ease the task.

Examples using Defaults and Flags:
On page 399 of *Data Structures Using Pascal* by A. M. Tenenbaum and M. J. Augenstein (1981, Prentice-Hall, N. J.), in the midst of a discussion of a *Heapsort* program, the authors say

"The last *if* statement reads

```
          if  j+1 ≤ k
              then if  x[j+1] > x[j]
                       then  j := j+1
```

rather than

```
          if  ( j+1 ≤ k )  and  ( x[j+1] > x[j] )
              then  j = j+1
```

because we must ensure that the references to x[j+1] and x[j] are within array bounds."

Since the references to x[j+1] and x[j] cannot lie beyond array bounds when j+1 ≤ k , the BOOLEAN expression (x[j+1] > x[j]) can be invalid (DTSTR) only when it doesn't matter; so the second statement makes perfect sense, seems simpler to understand, and would execute faster than the first if it were allowed to continue executing on those rare occasions when it is technically invalid. Only a Martinet could insist upon stopping computation instead of continuing with an exceptional value like NaN for x[k+1] .

Language purists might protest that assigning a meaning to the second statement above when the language Pascal specifies that it be undefined is a corruption of Pascal's semantics. They might offer a *conditional AND* construct as in the language C , wherein we could write

```
          if ( ((j+1) <= k) && (x[j+1] > x[j]) )  j++
```

to accomplish with syntax similar to the second Pascal statement what the first does. But syntax is not the issue. Regardless of language, a computer could run faster if it could overlap the evaluations of ((j+1) <= k) and (x[j+1] > x[j]) ; yet such overlap must be proscribed if the latter expression is capable of a side-effect like stopping computation. A conscientious C programmer, aware that the cautious user of his program might compile it with bounds-checking enabled if the user's compiler provides such a service, has to use the slower && operator; another shrewder programmer, reckoning that bounds-checking is not a standard feature of C , would use the unconditional & operator instead, and his program would run faster except when compiled by the cautious user. When the shrewder programmer's program aborts prematurely, who should take the blame and change his ways? Because the user and programmer disagree about that, an out-of-bounds reference to an array deserves to be classified as an exception rather than an error.

To forestall misunderstanding, let me repeat: I do *not* insist that execution *never* be aborted after an exception. Whether to abort or continue is a choice that I think belongs to the user or the programmer, not to petty tyrants who construct computers and compilers. Had language designers provided some syntax by which to distinguish between an array reference that will abort if out of bounds and one that won't, say x[...] versus x[<...>] , the choice might not fall into the realm of exception handling.

But, lacking that convenient mechanism for exercising a choice at compile-time, we are obliged to adopt a linguistically ugly mechanism that I call a *mode*, effective at execution time, that can be retrofitted into the run-time library of existing computer systems, in most cases without changing the compiler. To alter the mode of response to out-of-bounds array references x[...], a call will have to be made upon a library program that tells the exception handler associated with a DTSTR exception whether to ABORT or to DEFLT (DeFauLT). The latter mode supplies a value, possibly unpredictable, for attempts to read x[...] out of bounds, and ignores requests to write over it. This expedient seems no uglier to me than the extra-linguistic mode by which a Fortran 77 programmer tells the compiler whether to execute a zero-trip DO loop once (as in Fortran 66) or not.

Other language purists might wonder whether exception handling could be bypassed by writing programs like Heapsort in better ways free from extra tests and branches as well as from array references out of bounds. Yes, better versions of Heapsort do exist; and finding one makes a challenging exercise for students of Programming Style since most of them cannot find it unaided. (They get hung up on elementary inequalities.) But the exercise is pointless in an industrial setting where elegance is so often its own sole reward and never an excuse for slipping a schedule.

A Vectorizable Loop:

Similar considerations apply to the statement

```
    For  $k = 1$ to $N$  do
         if $(y_k/x_k > 3)$ and $(|x_k| > |y_k|^3)$ then  $z_k := \sqrt{z_k}$ ;
```

it is intended to replace z_k by $\sqrt{z_k}$ whenever (x_k, y_k) lies strictly inside a propeller-shaped region of the (x, y)-plane. What should it do to z_k if $x_k = y_k = 0$? The right thing to do is clearly to replace z_k by $\sqrt{z_k}$ rather than stop on an INVLD or ZOVRZ exception. The possibility that $z_k < 0$ adds a further complication; presumably this is not expected to occur when $\sqrt{z_k}$ is actually needed. On a vectorized computer with division and square root built into the hardware, the compiler would overlap the computations of $|y_k|^3$, y_k/x_k and $\sqrt{z_k}$ to create a BOOLEAN vector **b** with which to select the correct values for z thus:

```
   For  $k = 1$ to $N$   { in parallel }
      do begin      { overlapped }
         $b_k := (y_k/x_k > 3)$ and $(|x_k| > |y_k|^3)$ ;  $r_k := \sqrt{z_k}$ ;
         $z_k :=$ if $b_k$ then $r_k$ else $z_k$
      end { for k ... } .
```

How would that kind of overlap be accomplished if irrelevant INVLD, ZOVRZ or FODOM exceptions had to abort computation? A way that does not attempt .../0 nor √(negative) does exist:

```
   fodom := FALSE ;
   For  k = 1 to N  { in parallel }
      do begin      { overlapped }
           $bx_k$ := ($x_k$=0) ;   $bz_k$ := ($z_k$<0) ;
           $x1_k$ := if  $bx_k$  then  1  else  $x_k$ ;
           $y1_k$ := if  $bx_k$  then  1  else  $y_k$ ;
           $z1_k$ := if  $bz_k$  then  1  else  $z_k$ ;
           $b_k$ := ($y1_k/x1_k$ > 3) and (|$x_k$| > |$y_k$|3) ;   $r_k$ := $\sqrt{z1_k}$ ;
           $z_k$ := if  $b_k$  then  $r_k$  else  $z_k$ ;
           fodom := fodom or ($bz_k$ and $b_k$) ;
         end { for k ... } .
```

This program sets fodom := TRUE only if the previous program would have raised the FODOM flag, and then (bz_k and b_k) finds the values z_k that the previous program would have set to NaN .

The cumbersomeness of the last program is the price paid for a policy that aborts execution on DIVBZ, INVLD, ZOVRZ or FODOM; if OVFLO or UNFLO aborted too the price would rise beyond bearing. Continued execution, with flags raised when necessary, is a more economical policy. And if the programmer must know whether some of the elements of z are contaminated by $\sqrt{}$(negative), he can test the INVLD flag or, better, the FODOM flag afterward and then, only if it is raised, spend time re-examining z to find NaNs.

Solving an Equation:

The two examples so far showed how exception handling influences the way programs are written locally, near the site of potential exceptions. The next example shows how exception handling can affect the structure of a program globally, at a higher level; to appreciate the example's significance you must imagine how you would cope with similar examples on your favorite computer.

Consider solving for x the equation f(x) = 0 given an explicit expression for f(x) and a library of software, precompiled for your machine, from which to choose an equation solver. We shall call the solver [SOLV] because that is the key to press to solve an equation on an hp-28C calculator, and we hope to do as well on any other computer. Our program goes something like this:

```
MAIN program:
EXTERNAL REAL FUNCTION  f(REAL) ;
LIBRARY REAL FUNCTION  [SOLV]( REAL FUNCTION, REAL, ...) ;
...  Choose one or two initial guesses for x ;
x := [SOLV]( f, guessed_x, ...) ;
...  Use the solution  x ;
END of MAIN program.

REAL FUNCTION  f(REAL x):
f(x) := ln(x)*$\sqrt{}$(10 - x) ; { ... say }
END of  f .
```

Unlike the example chosen here, whose zeros (x = 1, x = 10) and domain ($0 < x \leq 10$) are obvious upon inspection, f(x) will often be an expression so complicated that its domain is practically inscrutable, though its application will suggest first guesses x inside the domain.

When our example is run on an hp-28C, first guesses x somewhat less than 5.42 yield the zero $x = 1$; bigger first guesses yield the zero $x = 10$. As long as one guess lies in the domain of f , the calculator always produces one of those zeros or the other. Other machines do less well. When a similar program is programmed in C and run on certain old UNIX™ systems that set $\sqrt{}$(negative) := 0 with ERRNO := EDOM , spurious zeros $x > 10$ are sometimes delivered with no warning except ERRNO (but who looks at that?); at other times overflow aborts computation of f(negative) . On other computer systems that abort $\sqrt{}$(negative) and ln(nonpositive), only first guesses close enough to $x = 1$ deliver that zero; the zero at $x = 10$ is inaccessible, and aborted computation is the reward for most guesses.

The hp-28C fares so well because its zero finder knows what to do when $f(x)$ is sampled at a point x outside its domain; look elsewhere for a zero. And computation of f is not aborted by invalid operations but continues with an exceptional value that will not deceive the solver. This is a special case of a policy good for search programs generally; such programs include ...

- Equation solvers that search for a zero,
- Optimizers that search for an extremum, and
- Query managers that search a data base for an answer.

Like any hunter, the search program must seek its quarry in places wherein may lurk something more dangerous than the quarry, something that can terminate the hunter instead of just the hunt. A policy that mitigates the danger is to continue execution after an exception provided either that the exceptional value (like a NaN) will not foist the wrong quarry upon the hunter, or that a deservedly raised flag will warn the hunter off a false scent but not deflect him from the true. Such a policy seems simpler than throwing and catching signals, setjumps and longjumps, ON ERROR statements and various other techniques that handle exceptions as interrupts; exceptional values and flags do not usurp a program's normal path of control. However, the next and last example in this section of the paper brings us back into a complicated world.

Vector norm:
The following example is included because it is traditional. We consider now a function subprogram norm(x) that calculates the Euclidean norm of a real vector $\mathbf{x} = (x_1, x_2, \ldots, x_N)^T$ from the formula norm(x) := $\sqrt{(\mathbf{x}^T\mathbf{x})}$ wherein $\mathbf{x}^T\mathbf{x} := x_1^2 + x_2^2 + \ldots + x_N^2$. Unlike most previous treatments of this problem, ours presumes nothing about the order in which the sum will be computed, allowing for the possibility that multiplications and additions will be overlapped on a pipelined or vectorized machine. Like all previous treatments, ours does attempt to circumvent the spurious over/underflows that occur on rare occasions when all of the x_ks are so big or so small that their squares over/underflow although norm(x) , if computed correctly, is unexceptionable. For that purpose certain machine-dependent constants are needed.

One constant is a scale factor h , the smallest power of the machine's radix whose square h^2 overflows. It is a power of the radix to ensure that multiplication and division by h are always exact (no rounding error) provided they do not over/underflow.

(This definition of h is ambiguous on CRAYs because their overflow threshold is ambiguous,- it depends upon the operation; so we pick the larger of two possible choices h .) The second constant is the difference eps between 1.0 and the machine's floating-point number next less than 1.0 . (But eps should be determined from the number of significant digits that floating-point numbers carry rather than by actual subtraction on those computers that lack a guard digit for subtraction; among such computers are CRAYs, CDC CYBERs and UNIVAC 11xx's . On a CDC CYBER that subtraction could produce zero instead of eps .) The third constant t is the largest floating-point number such that eps^2 t underflows to zero. (t is ambiguous too on CRAYs and CDC CYBERS because they can "partially underflow" but that does not matter.) All three constants can be computed from a function Nextafter(y, z) that returns the machine's floating-point number adjacent to y on the side towards z , provided subtraction is carried out with a guard digit as it is on IBM 370s, DEC VAXs, all machines that conform to IEEE standards 754 or 854, and a host of others; we shall compute the constants that way.

The default response to UNFLO is presumed to be continued computation with an exceptional value that is 0.0 or a subnormal number as specified by IEEE 754 and 854. If the machine lacks an UNFLO_flag, omit all references to it from the program. Also the default response to OVFLO is presumably continued computation with an exceptional value that is $\pm\infty$ or huge and a raised OVFLO flag. These presumptions are presumptuous because some machines just stop on OVFLO, and others continue without a flag; to compute norm(x) reliably on such machines is a problem left to the reader.

```
      ...
   eps := 1.0 - Nextafter(1.0, 0) ;
   radix := (Nextafter(1.0, ∞) - 1.0)/eps ;
   huge := Nextafter(∞, 0) ; ...  presumed > 1/eps^5 .
   h := radix^(integer no less than  0.5*ln(huge)/ln(radix) ) ;
   t := Nextafter(0.0, 1)/eps^2 ;
      ...
REAL FUNCTION norm( REAL VECTOR x) ;
   REAL s, c ;
   unflag := FFLAG(UNFLO, FALSE) ;  ... saves and lowers
   ovflag := FFLAG(OVFLO, FALSE) ;  ...   system flags.
   s := xᵀx ;  ...  over/underflow could happen here.
   ovflag := FFLAG(OVFLO, ovflag) ;  ... copied and restored.
   c := 1.0 ;  ... in case over/underflow didn't happen.
   IF ovflag THEN  c := 1/h  ...  x must be very big.
        ELSE IF  s < t  THEN  c := h/eps ;  ... if x is very tiny.
   IF  c ≠ 1.0  THEN  s := (c*x)ᵀ(c*x) ;
   ...  OVFLO cannot have occurred since its flag was restored.
   unflag := FFLAG(UNFLO, unflag) ;  ...  restored.
   RETURN  norm := (√s)/c ; ...  signals only as necessary.
   END ... norm.
```

This program runs about as fast as any program could run in the usual situation when neither overflow nor serious underflows occur to invalidate the first computation of s . Only when necessary does the program scale x and recompute s . Without flags, x would have to be scanned for its biggest element to decide whether scaling is needed; that is the time this program usually saves.

```
MORE TO COME LATER ABOUT ...
        Individual exceptions:
                 Unsupported exceptions.
                 Multiple exceptions.
                 Saving/restoring all flags at once.
        Modes :  ABORT    abort computation
                 PREMT    pre-empted by the language
                 DEFLT    Default mode  ( IEEE 754/854)
                 PAUSE    ..., look around,  and resume.
                 COUNT    over/underflows up and down.
                 PRSBS    Presubstitution.
        Scope :  with language help,  and  without.
                 localization of flags and modes.
                 special effects for leaf-subprograms.
                 simulation of atomic operations.
        Retrospective Diagnostics :
                 with language help,  and  without.
                 with operating system help,  and without.
                 with error-traceback,  and without.
                 with precise interrupts,  and without.
                 Flag-annunciator on console screen.
                 Circular  "Standard Error file"  on disk.
        Existing schemes :
                 Fortran  on  IBM 370s
                 APPLE's  SANE
                 etc.
```

See also "Presubstitution, and Continued Fractions" for examples where Presubstitution pays off.

David Barnett's "A Portable Floating-Point Environment" for partial implementations on a DEC Vax and a Sun III

"7094 II System Support for Numerical Analysis" in SHARE Secretarial Distribution SSD 159 (Dec. 1966), item C4537, pp. 1-54

Pat H. Sterbenz "Floating-Point Computation" ch. 2 (1974) Prentice-Hall, N. J.