CONCERNING SOFTWARE RELIABILITY

This note concerns the paper "Software Reliability" by Elspas, Green, and Levitt in the January/February, 1971, issue of Computer. The article is a worthy contribution on a very important subject. However, I disagree with a few statements made in the article, and accept your invitation ("Some Comments from Our Readers," p. 70 of the same issue) as a forum for discussion.

In example 3 (p. 23) the problem of erroneous data access caused by an index error is cited as a defect in the FORTRAN language. In contrast ALGOL 60 is cited as a language in which references to data outside of array bounds yield an undefined quantity. This is equally unsatisfactory, since every access using a subscript will require an additional execution time to test whether the subscript is in bounds. Perhaps the authors are aware of some technique for rapid indexing in which such tests are not necessary; if so, I would be very interested to learn. Otherwise, a reasonable solution would be a debug compilation, in which subscripts are tested before being used, with an optimized compilation for debugged programs. This solution applies to FORTRAN as well as ALGOL; there is no apparent advantage in providing an "undefined value" instead of a run-time diagnostic.

The authors are correct in concluding that semantic checking would be helpful in debugging programs. However this is an extremely delicate subject with dangerous pitfalls. An example is provided in this article—example 6. On the surface it appears that it would be an error in FORTRAN to write two statements which store a value into the same location with no intervening branches, statement labels, or references to the variable. On the contrary, the following example shows how it might reasonably occur.

It is often useful to override computations for debug purposes.

10 A = X**2 + Y**2
20 A = 500.0

Statement 20 is temporary. Its insertion permits the programmer to test the case when A equals 500.0 without having to manipulate the primary inputs in order to force this condition. After debug, statement 20 would be removed. This practice is obviously preferable to the temporary replacement of statement 20.

Admittedly a "non-fatal warning diagnostic" could be ignored in this case; however warning diagnostics are also dangerous when the programmer must sift through the unwelcome messages to find the important ones.

Defending FORTRAN places me in an unusual position, because I disapprove of the language for many of the same reasons expressed in the article. But FORTRAN is surviving in spite of the growth of sophisticated languages that embody some of the concepts described by Elspas et al. Programmers welcome these sophistication; but physicists, biologists, etc. comprise a substantial percentage of the population of computer users, and they are naturally unwilling to spend a great deal of time in learning to program. It is all too easy to forget that the ultimate objective of programming languages is not necessarily to please programmers.

Very truly yours,
Stephen N. Cole
IBM, Warminster, Pennsylvania.

The following comments are in response to S. N. Cole’s discussion of our article "Software Reliability."

Concerning the feasibility of run-time checks on array bounds, I became convinced of the value of this error detection facility while doing occasional ALGOL programming on a Burroughs B-500 machine which had built-in hardware implementation of the bound-checking feature. As Cole correctly points out, most contemporary machines cannot do this job efficiently because of hardware design limitations or inappropriate instruction sets. This is a pity because bound-checking can be accomplished efficiently in several ways by doing the index bound comparisons concurrently (in parallel) with the address modification step that computes \( \text{BASE} + \text{INDEX} \). For machines having a word length adequate to store the bounds and base in one cell the time penalty involved need not exceed one extra instruction fetch per level of indirection.

If a high-level language expects array-bound checking to occur at run-time and this cannot be efficiently managed on the "host" machine, then the alternative mentioned by Cole is a reasonable compromise. That is, compile the necessary software to make the checks during the debugging phase—and then recompile omitting these instructions for the production program. Another compromise is to use bound-checking on stores but not on fetches. The latter strategy permits nonsense calculations to occur but denies the capability of "smashing" data or instructions outside of referenced arrays. Both of the options mentioned above have been implemented in FORTRAN compilers.

With regard to the use of special number representations for undefined quantities, it is a handy debugging aid to be able to load a program with all storage locations set to an undefined number. If the arithmetic rules of the machine are such that all computations using an undefined number yield an undefined result, then some common errors such as failure to initialize a program variable show up immediately. Mistakes of this type are less easy to locate when storage is preset to "legal" numbers e.g., zero or "garbage" during the loading phase.

Considering the utilization of redundant assignment statements as illustrated by Cole’s example, one should note that while his code is semantically correct within the context of a debugging run, it would not be correct in the final program. That is why we explicitly stated (with reference to a similar example in our article) "the piece of program could conceivably be correct, although it would contain a redundant statement." I have used exactly the above artifice of Cole’s many times. On one occasion it cost me half a day of debugging a large program to discover that I had forgotten to remove one of the redundant patches. For this reason alone we would favor a non-fatal diagnostic message as stated in our paper.

As a final remark, the authors believe that the design of source-languages and the development of automatic program verification techniques are two areas in which improvement and innovation will tend to mechanize portions of the program creation process. Progress in these areas should eventually be of benefit to programmers of all levels of skill not merely the professionals.

Sincerely yours,
Milton W. Green
Senior Research Engineer
Computer Science Group
Stanford Research Institute
Menlo Park, California