Part of another document.

7.5 Scope

The different *modes* of exception handling discussed above can all be implemented as part of the same run-time support library as contains COS and EXP, with minimal interaction with the language or its compiler. Selection of a mode can then be treated like assignment of a value to a global variable. That approach has the virtue of easy implementation, but it also has two disagreeable properties. First, any subprogram that changes a mode must also almost always save it for restoration upon RETURN lest the calling program's mode be changed as an unintended side-effect of the subroutine. Second, treating the mode of exception-handling as a global variable is tantamount to making it an argument passed implicitly by reference to each subprogram; that subprogram may malfunction if it expects a mode (usually the default) different from the one actually in force when the subprogram was called. To treat modes as global variables is to forego too many advantages of block structure in languages that have it, reviving disputes reminiscent of whether the called program or the caller should be held responsible for saving and restoring scratch registers. Yet, for all its faults, this approach has been used successfully in the Standard Apple Numerical Environment (SANE) on Apple II and Macintosh computers; see the "Apple Numerics Manual" (1986) Addison-Wesley Publ. Co. SANE includes two library procedures *ProcEntry* and *ProcExit* intended to be called respectively at the beginning and near the end of a subprogram in order to save and restore modes, and hide or expose exceptions.

But exception-handling modes are rarely changed,-- not so rarely that they can be ignored, but too rarely to be remembered every time. Consequently, programmers may well forget to establish the correct mode when a subroutine is entered, or to restore *status quo ante* upon exit. Here is where a higher-level language should help diminish the incidence of error by enforcing its block structure upon exception handling wherever that may be done to advantage. The idea here is to define the scope of an exception-handling mode in a way that resembles the scope of a variable name's declaration in that block-structured language. In effect, the compiler could invoke procedures like *ProcEntry* and *ProcExit* automatically and implicitly wherever necessary, thereby easing the task of changing modes correctly in subprograms.

Software that handles its own exceptions automatically is not yet so abundant that experience can tell us unerringly what scoping rules work best for exception-handling modes. Limited experience suggests that *ProcEntry* and *ProcExit* too often perform redundant work because the modes saved and restored are so often the same as those instituted by *ProcEntry*, namely the default modes. This is what we should expect when the default modes have been chosen well. Those few programs that do change modes do so infrequently; moreover, most of them seem to lie near either the root or the leaves of the subroutine calling tree. Programs near the root change modes usually to implement a *Policy*, like *Pausing* at every overflow in order to scrutinize it while debugging. Near the leaves, programs that change modes do so to hide irrelevant

exceptions, handle others specially, and expose the rest; hence they almost always restore the modes in effect when they were called. Since most exceptions are encountered first near the leaves, handling them there in ways prescribed by earlier mode specifications is the nicest strategy, but not always feasible. Another strategy, appropriate to the middle levels of the tree of subroutines, is to try an obvious algorithm first and find out later whether it worked; if not, amend it in accordance with the circumstances revealed by the exception that thwarted the first try. This latter strategy can be automated with the aid of tests of flags in suitably augmented codes; but usually the strategy is carried out by programmers who, if aware at all that exceptions might occur, prefer to wait until an exception handled improperly by the default response actually does occur before taking further steps.

If the foregoing strategies deserve to be enshrined in programming languages, a reasonable way to do so may be borrowed from APL's practice of "localizing system variables;" see APL*Plus, by STSC Inc., Rockville, MD 20852. This system has certain *system variables*, like *CT* (Comparison Tolerance) and *IO* (Indexing Origin), that programmers may change, within limits, as if they were ordinary global variables. However, a user-defined function (subroutine) may *localize* a system variable by including its name among the names of other local variables in the function's header (first line of declarations). So localized, the system variable will inherit its value within the function from the value it had outside, and revert to that value upon exit from the function. In effect, this passes a system variable by value (copied), instead of by reference, to the function.

Treating exception-handling modes like localizable system variables seems like a good idea. For instance, suppose FOVFMODE is a system variable that tells how floating-point overflow will be handled; allowed values might be ABORT, PAUSE, DEFAULT (to +∞), PREEMPTED, COUNTING, PRESUBSTITUTED, ... In the absence of any declaration to the contrary, a subroutine can treat FOVFMODE like any other global variable restricted by strong typing rules to a small set of values. A subroutine that declares FOVFMODE to be a local variable receives it by value, available for change, with the understanding that FOVFMODE will revert to its initial value upon exit. Moreover, the floating-point overflow flag will be saved and cleared upon entry, and merged with the saved value upon exit; and if the merged value differs from the initial value then an overflow will be signalled in accordance with the initial (and now restored) value of FOVFMODE. (This is how *ProcExit* acts.)

More could be said here about how to interpret mode assignments that look like CONSTANT or PARAMETER or DATA declarations in such a way as will save unnecessary calls to system subroutines that save and restore modes. More could be said about how to cope with imprecise interrupts and other impediments to exception handling in concurrent, vectorized, pipelined and parallel computation.

More could be said about resolving the incompatibility of pre-existing computer architectures of long standing with the needs of numerical software addressed by the IEEE standard. But this discussion has already gone far enough to accomplish its purpose. That is to sensitize compiler writers to the existence of a family of problems, concerned with floating-point exception handling, for which universally satisfactory solutions have yet to be promulgated. We do know that satisfactory solutions must involve (minimally, we hope) compiler writers.