SIX LANGUAGE EXTENSIONS TO ENHANCE THE PORTABILITY OF MATHEMATICAL SOFTWARE WRITTEN IN PL/I: BACKGROUND AND JUSTIFICATION*

by

Kenneth W. Dritz

Applied Mathematics Division

May 1982

*This work was supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research, U. S. Department of Energy.
# TABLE OF CONTENTS

Abstract .............................................................................................................. 5

1. Characterization of mathematical software ................................................. 6

2. Shortcomings of PL/I for mathematical software ................................. 9

3. Six extensions to PL/I for mathematical software ................................. 13
   3.1. Proposed features ............................................................................... 13
       (#1) Environmental enquiry functions ................................................. 13
       (#2) Generalization of restricted expressions ............................... 17
       (#3) Liberalization of the contexts of restricted expressions ........ 18
       (#4) Named-literal declaration type .................................................. 19
       (#5) Explicit precision specification for constants ...................... 24
       (#6) REQUIRE statement (a pragma) ........................................... 25
   3.2. Example ............................................................................................... 26
   3.3. Relationship to a preprocessor ......................................................... 37

Acknowledgments .............................................................................................. 39

References ......................................................................................................... 39
This report proposes six extensions to ANS PL/I, which is being revised by the American National Standards Committee X3J1.

The new features include environmental enquiry functions, generalization of "restricted expressions" (compile-time expressions), liberalization of the contexts of restricted expressions, a named-literal declaration type, explicit precision specification for constants, and a pragmatic statement for expressing conditions that an implementation must satisfy for acceptable compilation. Used together, these features will give numerical analysts access to properties of an implementation's floating-point arithmetic in exactly the ways required to ease the burden of tailoring a program's precision specifications to new environments. In many cases it will be possible to write PL/I programs that are completely self-adapting to their host environment.

Effective definition of the environmental enquiry functions will require the incorporation of an explicitly parameterized model of floating-point arithmetic. If such a model is integrated into the Standard, numerical analysts will be able to state and prove theorems about their programs' error bounds by appealing directly to the Standard.
This report represents an attempt to set forth, for language implementors who are not numerical analysts, some of the special properties of mathematical software, the shortcomings of PL/I as a language for writing portable mathematical software, and six tentative ideas for extending PL/I to serve that application area more effectively and competitively. This is not a finished, concrete proposal to that end. Originally circulated as ANSC X3J1 document 1981-32, it is intended to stimulate thinking and lay the groundwork for proposals.

A word about my use of the term "portable" is in order. In some circles, the term "portable" has acquired a precise meaning which is to describe source programs that, with no changes, work as intended on a variety of different computer systems. Throughout this report I use it more informally to mean usable in many environments (perhaps with some simple changes).

1. Characterization of mathematical software

Mathematical software usually refers to purely computational ("numerical") subroutines that compute some standard mathematical functions, or solve problems like ODEs and linear systems, of utility in a wide variety of scientific applications. Such software uses almost exclusively the binary floating-point and binary integer data types and generally avoids I/O. Other characteristics are discussed later.

Who writes mathematical software, and who uses it? A small community of numerical analysts is responsible for most of the high-quality mathematical software produced. Compared to the well-populated ranks of applications programmers, this group is indeed small, seemingly hardly large enough to justify the existence of language features dedicated to the solution of its particular problems. But the group is certainly a significant force in computing. Its products are widely distributed (in some cases by concerns that engage exclusively in writing, testing, and marketing mathematical software) and are commonly incorporated into applications programs. (John R. Rice of Purdue University has estimated that numerical computation accounts for 50% of the computing expenditures in the United States [13].) A prevailing view in the numerical analysis community is that its software provides semantic extensions to high-level languages, including "operators" for matrix
operations, equation solving, special function evaluation, etc., by the ubiquitous and trivial syntax of the CALL statement -- a view that is not without its merits. The importance of mathematical software is further underscored by the support of academic and government research in numerical methods, by college curricula in numerical analysis, by the existence of special interest groups like ACM SIGNUM, by the fecundity of authors in journals such as TOMS and the SIAM Journal on Numerical Analysis and in books on numerical analysis topics, and by the frequent symposia and workshops devoted to the field.

Two major goals of mathematical software writers are accuracy and speed. The former is of particular scientific interest because of the well-known pitfalls of trying to represent and compute continuous functions with discrete computing elements. It is often possible, with specialized methods developed by numerical analysts, to do significantly better than by evaluating textbook formulas. The numerical analysis community has accepted the responsibility to provide the scientific programmer with better computational kernels than he could write himself, and commercial and professional competition sustain the challenge of providing the best. Speed, too, is of course important; it is likewise a legitimate scientific concern of numerical analysts, because tradeoffs between speed and accuracy frequently have to be confronted.

Robustness is another desirable property of mathematical software. If the software is to be safely used in ways that are clearly beyond the control of its authors, it must be resilient to abuse. Sometimes, designed-in protection from inaccuracies in compilers is also called for.

To achieve their goals, designers of mathematical software must take into account various properties of the floating-point arithmetic unit of the target machine. These properties include significand length, hardware radix, the rounding mode, and the presence or absence of guard digits. And, especially in languages that provide no way to detect and recover from exceptional conditions such as overflow, adequate robustness and reliability can be achieved only when account is properly taken of such hardware limits as the overflow or underflow threshold, or the exponent range. Even in languages like PL/I which provide exceptional-condition handling, there are cogent
reasons why one would like to have access to these limits; some of these will be discussed later.

Clearly, with knowledge of particular hardware one may exploit any of the properties just mentioned to the desired level by programming in the assembly language of that hardware. That is also the obvious recipe for achieving the ultimate in speed. Formerly, numerical analysts did just that, but the proliferation of different architectures and the demand for mathematical software on a wide variety of machines -- added to economies in development and maintenance activities available when standardized and widely available high-level languages are exploited for portability -- have conspired to make that practice no longer cost-effective.

Some compromise, however, is to be expected when high-level languages are used. For example, because control over the precise code sequences generated is not possible, some speed is usually sacrificed. More importantly, high-level languages have not until recently provided access to the machine-dependent parameters which numerical analysts need. But current research in high-level language features that enhance the portability of mathematical software has brought us to the brink of an era in which traditionally incompatible goals of speed, accuracy, reliability, robustness, and portability may be simultaneously achievable to a greater degree than ever before.

One of the historical advantages of high-level languages is that they make possible machine-independent programming. It might well be asked, then, whether the features described above, and discussed in a PL/I context more fully below, would conflict with that advantage by encouraging machine-dependent programming. After all, numerical analysts are strongly hardware-oriented, and it must be acknowledged that they want to write programs to drive the hardware in particular ways. In a very true sense, their brand of programming is every bit as much a kind of systems programming as that of writers of operating systems and utilities [8]; it shares the need to capitalize on hardware characteristics and really differs only in its subject area.

The resolution of this apparent conflict lies in a proper understanding of what will be achievable when these new features become widely available. Their raison d'être is not simply to provide access to and exploitation of
machine characteristics, but rather to enable portable programs to adapt their behavior to the underlying hardware characteristics so as to achieve their specifications (of accuracy, reliability, etc.) on a broad class of different machines. The emphasis is squarely on portability. Given that, such programs are in reality machine-independent because, over a wide variety of different machines, they adapt their behavior to the host environment in just those ways needed to achieve uniform specifications relative to the whole set of machines.

2. Shortcomings of PL/I for mathematical software

It is to be expected that most programmers, particularly applications programmers, are concerned only with the details and requirements of their own programming environment. It is not surprising, then, that the problems of designing programs capable of being used in a wide variety of environments are not universally appreciated. In particular, trying to write portable mathematical software in PL/I can be an unexpectedly frustrating yet highly educational experience, one that is recommended to anyone who insists that PL/I is adequate.

A function-procedure designer's very first problem will likely be the selection of precision attributes for the parameters and returned value. If one were designing a procedure to be used in a single environment, one would try to guess the most likely precision that the applications programmer would employ throughout his program. More often than not this precision will relate to certain hardware characteristics, such as the maximum precision for the local single-precision floating-point data type. Matching that precision at the procedure interface is intended to avoid the inefficiency of dummy argument creation ordained by the language when the precisions of argument and parameter do not match, or alternatively to avoid forcing an "unnatural" precision on the user of the procedure. It is clearly not possible to solve this problem for a wide variety of different environments simultaneously by

In the case of an "output argument", creation of a dummy argument has consequences more serious than inefficiency.
picking a single number for the precision attribute. If he has not given up at this point, the designer of a portable procedure must either compromise some goals or seek ways to write the program so that it can be adapted easily by each potential user to that user's environment. The first of these alternatives is intolerable to numerical analysts and has contributed to their low regard for PL/I; the second is acceptable but not easily practiced without the new features proposed in the next section.

To solve the particular problem of argument/parameter matching in portable programs, Fortran seems generally better adapted than PL/I. In Fortran it is possible to say REAL (or DOUBLE PRECISION) for both argument and parameter, thereby achieving a match in any environment. Numerical analysts are used to the fact that different machines yield results of different precision for a given data type. Occasionally that is a problem, as when single precision is adequate for a particular solution method on some machines but inadequate on others possessing an unusually short word length. Proposals to address such problems are before ANSC X3J3 [1] and the IEEE Floating-Point Working Group Microprocessor Standards Committee [10].

One might attempt to achieve portability in PL/I by letting the precision attribute be supplied, on both sides of the procedure interface, by the default process. This is not an especially attractive solution because, for floating-point variables of a given base, only one default precision is distinguished by the language, in contrast to the two "defaults" of Fortran. (Also, that one default is given no hardware interpretation by the PL/I Standard [2].)

The DEFAULT statement of PL/I is potentially useful for achieving portability through easy adaptation of a program, by hand, to a particular environment. One could, for instance, write a pair of default specifications for single and double precision, which has the advantage that the "magic numbers" that correspond to these hardware data types are localized in the program and easily changed. The two defaults could be applied to disjoint sets of variables by adapting some naming conventions and using the RANGE predicate in the DEFAULT statement. But is this good enough?

Unfortunately, no. The problem with differential defaulting by naming conventions is that some language elements which have (or involve) a precision attribute, and can indeed acquire it by the default process, do not have
names. Thus, while it is possible to write a default specification which gives constants not protected by the default-suppression character a particular precision independent of the number of digits in their textual representations, it is not possible to default some REAL FLOAT constants to one precision and some to another. Nor is it possible to default some parameter-descriptors or returns-descriptors to one precision and some to another, unless the two classes differ in at least one other attribute.

The defaulting process fails in yet other ways to provide a means of localizing precision specifications in a program. A number of syntactic constructs involve precision specifications in some form but do not interact with the defaulting process at all. One area of the language exhibiting this problem is the set of arithmetic built-in functions. Most of these take one or two arguments (which must be expressed as decimal integer constants) denoting the precision of the result. In certain circumstances these arguments may be omitted, but then the implied precision is determined by a process other than the uniform defaulting process. These built-in functions are potentially of special interest to writers of mathematical software because of the control over the precision of operations which is achievable by their use. But, if they are extensively used in a program, absolute precision specifications will be dispersed throughout the program, resulting in one that is not easily adapted to other environments. (Another language feature that suffers the same problem is the so-called "generic-precision", i.e., the precision-range specification which may be written in a generic-selection predicate; ironically, this, too, is a feature intended to ease the burden of writing machine-independent numerical programs but which, in some ways, interferes with that very goal.)

Several problems related to the specification of explicit precisions (that is, problems having nothing to do with inadequacies of the defaulting process) may also be cited. One of these has to do with arithmetic constants. Though it is possible to write such constants with excess precision and have them all interpreted in accordance with some "working precision" provided by an appropriate default specification, it is sometimes necessary to single out a few individual constants to have some other explicitly stated precision; but there is simply no way to do that independently of choosing the
number of digits to write. A technique is needed to specify the precision of a constant independently of the way it is written, so that individual constants may be written with excess precision and then tailored to particular environments without deleting digits.

Another problem relating to explicitly stated attributes is that typically many different attributes (particularly precisions and string lengths) of different variables will be related through certain algorithmic or linguistic dependencies, and yet there is no way to capture those dependencies so that a change in one attribute causes the automatic adjustment of all the related ones. This effect is particularly noticeable when variables of different data types are employed in expressions or assignments, thus resulting in data conversion; often, little or no latitude is permissible in the choice of attributes if exceptional conditions or unexpected results are to be avoided.

One final problem having to do with explicitly stated precisions involves the inability to write two different floating-point precision specifications such that one is guaranteed to yield higher actual precision (in terms of the hardware representation chosen) than the other in all implementations. It is thus not possible to write a portable PL/I program that computes a quantity in some particular precision and a residual, or a correction term, in a (guaranteed) higher precision. This capability is crucial to many successful numerical methods. Also, in certain kinds of problems a given quantity is first computed in some "trial" precision and then recomputed in a higher precision to see if the results are improved (often this practice is employed in programs to test mathematical software, and there is a need to make such test programs portable). Because requesting higher precision does not always result in the use of higher-precision hardware representations, there exists the possibility of drawing the wrong conclusions when no further improvement is obtained.

The extensions outlined in the next section are intended to alleviate these shortcomings by increasing the control over precision specifications where that control is currently inadequate; by allowing the dependencies among various specifications in a program to be captured and made explicit in expressions, so that hand-tailoring of a program to a new environment can be
made more reliable (and kept to a minimum); and by providing a way for the basic, independent specifications of a program to be linked automatically to parameters of the environment, which, in conjunction with the other extensions, offers the prospect of eliminating hand-tailoring entirely in many cases.

Certain features desired by the numerical analysis community are addressed in this report. In particular, I do not consider here the requirement for an ON-unit for the overflow or underflow condition to be able to receive information about the exceptional value and deliver to the point of interrupt some computed alternate result [10].

3. Six extensions to PL/I for mathematical software

This section is organized as follows: first, a presentation of current ideas on the extensions; second, an elaborate example of their use; and third, a discussion of the relationship between these extensions and a standardized preprocessor for PL/I.

3.1. Proposed features

The six features described here are closely related and meant to be used together. Some of them will not have much use without the others. Thus, they should be considered as a package.

(#1) Environmental enquiry functions

These are a set of built-in functions that return certain key properties of the host environment. Some of these functions will have no arguments, while others will take arguments. The returned values will have data types that in some cases are associated only with the name of the built-in function and in others are related in some way to its arguments. In each instance the value of an environmental enquiry function is implementation-defined.

The first statement of the advantage of having machine-dependent parameters available to high-level languages as environmental enquiry functions was made by Naur in 1967 [12]. Since then, the concept has been refined and
related to the idea of a model for floating-point computation [4], an
abstraction to which the behavior of a real computer can be made to fit by
specifying as parameters of the model certain actual properties of the
and have been proposed by ANSC X3J3 [1] for the next Fortran standard.
Numerous groups have advanced their favorite proposals for environmental
enquiry functions. A recent discussion of the history and current state of
these developments, including a comparison of the leading proposals and an
extensive bibliography, has been written by Cody [5].

What environmental enquiries are needed? Much more work is needed to
decide which ones are absolutely necessary. At the present time I can only
mention a few and perhaps hint at several classes of others. Later in this
section some concrete examples will be presented.

Numerical analysts frequently need to know the hardware radix and hard-
ware precision (significand length in terms of radix-digits) used for
floating-point variables; the largest and smallest positive floating-point
quantities; and "epsilon", the "relative precision" or "maximum relative
spacing", which is often defined as the smallest positive floating-point value
that, when added to unity, produces a quantity different from unity. It is
sometimes also desirable to know whether the machine chops or rounds, whether
it has guard digits, etc.

The actual values of certain of these parameters will of course differ
depending on whether one is talking about single-precision or double-precision
hardware, and they might also differ with the base if the hardware happens to
have both binary and decimal native floating-point representations. Thus,
these environmental enquiries are functions of data types. In Ada, where
types can be treated in limited ways like values, that fact is quite clear.
In the proposal for Fortran, the type in question is that of the variable
presented as an argument to the environmental enquiry function; the value of
that variable is of no concern. As will be seen shortly, certain environmen-
tal enquiries are essential to have available in particular compile-time con-
texts. For that reason and others, the Fortran means of relating an enquiry
to a type is not possible in PL/I. PL/I also lacks the general mechanisms
required to support the Ada approach. I therefore propose that the base to
which the enquiry applies be a part of the name of the enquiry and the
precision be an integer argument. Thus, for instance, the value of
EPSILON2(45)\textsuperscript{+} would be (informally) the smallest positive value of \( E \) for which
\( X+E \) differs from \( X \) where \( X \) has the value unity and \( X \) and \( E \) are declared REAL
FLOAT BINARY PRECISION (45); the attributes of the reference EPSILON2(45) are
also REAL FLOAT BINARY PRECISION (45). Note that since the precision of
EPSILON2 depends on the value of its argument, that argument (like the
"precision" arguments of FIXED, FLOAT, etc.) must be evaluable at compilation
time. This requirement is shared by the precision arguments of all of the
environmental enquiry functions.

Defined in this way, the environmental enquiries could be supported in
"restricted expressions" (compile-time expressions). Those whose value is
interpreted as a precision could be used in numerous contexts, including the
PRECISION attribute itself, where a precision is required but must currently
be expressed by a decimal integer constant. This capability is essential to
the efficient implementation of software certain of whose specifications
automatically adapt themselves to the host environment, as the example will
show. Thus, if the value of ACTUAL_PRECISION2(n) is (informally) the largest
value of \( k \) such that REAL FLOAT BINARY PRECISION (n) and REAL FLOAT BINARY
PRECISION (k) variables have the same hardware representation, then use of
ACTUAL_PRECISION2(1) in a PRECISION attribute affords a portable and reliable
way of requesting single precision (if the hardware provides such), while
ACTUAL_PRECISION2(ACTUAL_PRECISION2(1)+1) serves in the same way for double
precision (if the hardware provides such). This particular environmental
enquiry function also offers a way of guaranteeing that the number of digits
accommodated by the internal representation of one variable (or one compu-
tation) exceeds that of another by at least so many digits. Finally, functions
such as MAXIMUM_PRECISION2() and even DEFAULT_PRECISION2() have uses that are
not difficult to recognize (when these function references can be used in
PRECISION attributes or the precision arguments of the arithmetic built-in
functions).

\textsuperscript{+}Throughout, the suffixes 2 and 10 are used in naming environmental enquiry
functions that have separate versions for binary and decimal base, even though
this differs from the rationale for the existing built-in function names LOG2
and LOG10.
While the enquiries that return the minimum and maximum floating-point values (of a given base and precision) might be used to avoid underflow or overflow, they have other (more essential) uses, such as in rescaling a problem after underflow or overflow occurs and in preparing portable test programs.

The existing built-in functions COLLATE, HIGH, and LOW are environmental enquiry functions in the true sense of the word; their use, however, lies in character manipulation applications rather than in mathematical software.

Closely related to the environmental enquiry functions is a class of built-in functions, called environmental manipulation functions, which are equally important to numerical analysts. Whereas the enquiry functions are static parameters of the host environment and are thus usable in compile-time expressions and in attributes, the manipulation functions depend on or do things to the values of variables and thus have meaning only during the execution of a program. For numerical work, manipulation functions to extract the unbiased exponent or the fraction of a floating-point variable, or to set the exponent or fraction, or to scale by a power of the radix (by incrementing or decrementing the exponent) are essential. These are typically used in argument reduction schemes and in function value composition to perform certain operations without roundoff error.

Environmental manipulation functions for numerical analysis can be understood on two different planes. On the one hand, they appear to be very closely related to the hardware representations of floating-point values. (This view tends to support the characterization of mathematical software as numerical systems programming.) On the other hand, they can be defined in terms of their essential mathematical properties [3], thus emphasizing their more abstract aspects. The hardware orientation is important only insofar as it suggests accurate and efficient implementations of the functions.

PL/I already has one environmental manipulation function: UNSPEC. It is clearly the most basic and primitive of all. Thus it can be used to perform all of the operations on floating-point variables described above. However, its use for that purpose requires knowledge of such implementation parameters as the exponent width, the significand width, the normalization mode, the exponent bias, and the sign representation, and it is thus unsuitable for achieving portability. The higher-level functions proposed for extraction,
setting, and scaling hide these details and thereby permit a much greater degree of portability to be achieved.

How shall these enquiry and manipulation functions be defined in the Standard if they clearly must depend on properties of the implementation? The key is to recognize the relationships among the different functions and exploit them as much as possible. If this is done properly, a few key implementation-defined parameters will emerge as the independent parameters of the underlying model of arithmetic, and everything will turn out to be definable in terms of these and derived quantities. The consequences of applying this approach can and should be carried to the extreme of relating the results of arithmetic operations to the parameters of the model, thereby establishing an implicit link between the environmental functions and the implementation's arithmetic [9]. The definition of PL/I can be made "tighter" and more effective and useful by this technique. For example, the conditions under which underflow or overflow may be raised by an operation, instead of yielding a value, can be stated in terms of appropriate model parameters; this closely ties the values returned by the environmental enquiries for the minimum and maximum floating-point values to the underflow and overflow thresholds. The bulk of the effort to elaborate the details of these numerical analysis extensions will be expended on identifying the needed environmental functions, retrofitting an effective but sufficiently broad model of floating-point arithmetic, and checking that all the operational consequences of the newly defined functions and constraints are internally consistent.

(#2) Generalization of restricted expressions

It is essential that the new environmental enquiry functions be usable in moderately general ways at compilation time. The example later will show numerous instances of this and of the occurrence of some of the existing mathematical and arithmetic built-in functions in restricted expressions. I propose that restricted expressions be composable of constants, named literals (described later), named constants (i.e., files, entries, formats, and labels), and references to any built-in function whose arguments are restricted expressions, as well as all meaningful operators and parentheses. No syntactic extensions are required and, in fact, I believe the required changes
in the Standard will be negligible. The cost to implementors will not be negligible, but the benefits to users (as part of this total package) will be substantial.

(#3) Liberalization of the contexts of restricted expressions

The concept of a restricted expression applies in contexts where the full generality of expressions is meaningless. In such contexts, basically in declarations of STATIC variables and in parameter-descriptors and return-descriptors, expressions are restricted to contain only a few kinds of operands (that can be evaluated at compilation time) and a few operators. The preceding extension was concerned with enlarging the content of restricted expressions; this one is concerned with syntactic extensions that will allow expressions in certain contexts, where only decimal integer constants are now permitted. All of these are contexts where a number is required during translation, so the expressions must be restricted expressions. Specifically, the following contexts are essential: the PRECISION attribute and all the other attributes containing a precision, including a "generic-precision" in a generic-description; and the precision arguments of the arithmetic built-in functions (e.g., ADD, FIXED, PRECISION, ROUND, and their brethren) and of the environmental enquiry functions. An additional essential context is presented later with extension #5. Other potentially useful contexts may be identified but they are nonessential.

The adoption of this feature carries with it certain costs for the implementor and risks for the user. The translation of an arithmetic constant becomes potentially recursive and, in fact, an infinite regression can occur unless the programmer takes steps to prevent it or the compiler recognizes the danger. For, during the translation of a constant, a default specification could be applied (as at present) to supply a precision. The translation and evaluation (as a restricted expression) of an expression in the precision default may require the translation and evaluation of further constants, which will require further default processing, etc. For example, the simple statement

DEFAULT (CONSTANT) PRECISION (8);

produces an infinite regression once the interpretation of "8" as a full-fledged constant is permitted. The alert programmer can prevent it by
suppressing the application of defaults to constants in precision defaults; e.g., he would write in this case

\[
\text{DEFAULT (CONSTANT) PRECISION (8P)};
\]

I do not believe these dangers warrant the dismissal of this feature, a key element toward the provision of a capability for expressing portable precision specifications. If additional "protection" of the programmer and simplification for the implementor are deemed desirable, other steps, none of them very attractive, could be taken.

It has been suggested that restricted expressions be permitted inside parenthesized replication factors in pictures. This would allow the length and certain details of picture specifications to be made parametric and dependent on the environment. Unfortunately, however, these replication factors are enclosed in the quotes that surround pictures, and the requirement to parse an expression enclosed in quotes is rather unappealing.

(#4) Named-literal declaration type

The need to associate a name with an arithmetic literal (i.e., self-denoting constant) has been perceived in the past. The usual justifications cited for this feature are its value as a documentation aid and the increased reliability possible when a complicated constant needed in numerous places can be referenced symbolically. An additional justification is the efficiency gained by providing a means, as is done here, to force the evaluation of "constant expressions" at compilation time. STATIC INITIAL variables have been used in the past to achieve all these advantages. A named literal differs from a variable declared STATIC INITIAL semantically: the compiler knows that the value of a named literal cannot change, so it may be able to optimize better and can at least diagnose as errors attempts to assign to one. Finally, because of the way in which the named-literal feature is integrated with the rest of these extensions, all of these benefits will be extended to the specifications (i.e., declarations) of a program.

The proposal, very tentatively, is to allow

\[
\text{DECLARE identifier LITERAL (restricted-expression)};
\]

wherein no additional attributes except INTERNAL can be written. (The same rule applies to names declared BUILTIN.) Effectively, "named literal" is
elevated to a new "declaration type" in apposition to "variable", "named constant", "built-in", "condition", and "generic". Note that the value is given by a restricted expression and is not constrained to be simply a constant. The declared identifier can be used anywhere a constant is permitted by the syntax and has the value and attributes of the restricted expression. In particular, the identifier can be used (like ordinary constants) in restricted expressions.

It seems desirable to me to exclude data-type attributes from a named-literal declaration. Attributes of the identifier are supplied entirely by attribute synthesis of the restricted expression and are affected by the application of defaults to any constants contained therein. Therefore, the attributes of a reference to a named literal will necessarily be exactly the same as those of the defining restricted expression, had that been written in place of the name. This seems to have definite conceptual advantages and incidentally happens to be the approach adopted in Ada (but it differs from the approach taken by Fortran 77 with regard to the PARAMETER statement). It also underscores the difference between LITERAL and STATIC INITIAL. In rare cases where "strange" attributes are needed for some named literal, it should be possible to force them out of the attribute-synthesis process, for example by using the capability provided by extension 85 or by composing appropriate built-in functions in the defining restricted expression. (The reader should begin to appreciate the interrelationships among the various parts of this package of extensions.) The prohibition against other attributes in a named-literal declaration helps also to clarify that a named literal, like other value references, does not have an alignment or a storage class.

It also excludes aggregates of named literals, at least for the present. Despite the enthusiasm shown in some quarters for allowing (for example) arrays of named literals, my feeling is that the problems to be solved before that can become a reality are grave. Also, I believe there is misunderstanding about the benefits of aggregates of named literals.

At least three problems with aggregate named literals can be identified. Firstly, aggregate value references currently are handled inconsistently, and it seems pointless to make things worse rather than better. For example, functions can return arrays or structures, while named constants (files, labels, formats, and entries) can only be aggregated into arrays, and literals
(computational constants) must be scalars; named-constant references can be subscripted, while function references can be neither subscripted nor "componented"; references to arrays of named constants are permitted, but not in restricted expressions (which must currently be scalar-valued); and subscripted (scalar) references to named constants are permitted in restricted expressions, such as initial value lists for STATIC variables. Secondly, an array of named literals necessitates an additional conversion of all the restricted expressions to their common derived type; this is undesirable for reasons to be discussed shortly. Finally, the strict analogy between named literals and unnamed literals is lost unless a syntactical device for the aggregation of the latter is designed (one which, furthermore, permits subscripting). Then, full exploitation of the generalization of restricted expressions to encompass aggregate values will demand a rethinking of the scalar orientation of initial-value lists, a subject that seems likely to be addressed (perhaps prematurely) by forthcoming proposals.

Some of the benefits of named literals may not be available for arrays of them, and this fact must be weighed when the utility of such arrays is assessed. In particular, if an array of named literals needs to be subscripted in the program by a variable, it will certainly need to be assigned storage just like a STATIC INITIAL array. Some optimization benefits are lost. The documentation benefit remains, but that benefit could have been provided by a STATIC INITIAL array. The diagnostic benefit (checking that the array is not altered) remains as the sole unique benefit.

One consequence of the prohibition against attributes in a named-literal declaration which is of considerable importance to numerical analysts is the elimination of a potential roundoff error when the value is converted from the attributes of the restricted expression to the declared attributes, should they be allowed. Should they be permitted in the declaration, the additional potential for roundoff error means that a constant as the sole component of the defining restricted expression will be subject to two such errors instead of one. The only remedy available to the programmer who needs to minimize the impact of that second conversion is redundancy, that is, arranging for the constant and the named literal itself to acquire the same attributes, including precision, either explicitly or by default. (The inelegance of redundancy is multiplied n-fold for arrays of named literals; furthermore, even with
attributes not permitted in named-literal declarations, a second conversion --
to the common derived attributes of all the restricted expressions -- seems
unavoidable in the case of these arrays.) In yet another situation, where the
restricted expression is a decimal integer constant and the named literal is
to be used in precision attributes and related contexts, the need to think
about attributes for the named literal is again unattractive; a program that
names the constants used in these contexts should, I believe, be as simple to
write as one that doesn’t.

Another question about which some controversy is likely to arise is
whether the defining restricted expressions for named literals should be
permitted to reference named literals declared earlier in the same block. To
do so imposes an order requirement on the processing of named-literal decla-
ration (which, in turn, legislates against allowing LITERAL to be acquired by
default). The example presented later assumes this capability and demon-
strates the convenience and conciseness that follows from it. Of course, the
capability is functionally nonessential, because the defining restricted ex-
pression for a named literal declared earlier can always be substituted† for
the named-literal reference in the later declaration. But in many cases that
will result in extremely lengthy restricted expressions in such contexts, with
repetitious occurrences of common subexpressions; in the extreme, it will have
the undesirable effect of necessitating the replication of “magic numbers” for
things like “working precisions” which are tailored to the host implementa-
tion. But to reduce that replication is precisely the reason for introducing
named literals. If that benefit is enabled for the body of a program and for
its ordinary declarations, but not for its named-literal declarations, a
defect in the rules governing the latter is implicated. I therefore strongly
advocate permitting such references and imposing a “sequential elaboration”
order on the named-literal declarations of a block. (Fortran 77, in its
PARAMETER statement, and Ada have this capability.) Justification for this
sequential elaboration rule can be sought from the interpretation of a named-
literal declaration as something closer to a compile-time assignment (of value
and attributes) than to a declaration, an assignment integrated into the

†However, obtaining identical results upon unfolding named-literal references
requires, even in this context, that attributes be prohibited in named-literal
declarations.
translation of a block at an appropriate point relative to declaration and
default processing. If additional precedent for the sequential behavior of
specifications is sought, one need look no farther than the DEFAULT statements
of a program.

The order in the Standard in which declarations and their components are
processed and defaults applied is already intricate, because of the need to
allow expressions in declarations to reference the names of built-in functions
declared in the same block and because of the requirements of generic selec-
tion during the translation of expressions in declarations. (For these
reasons, translation of expressions in the declarations of a block is deferred
until attribute keyword sets are completed for those declarations.) The
interaction of the translation of named-literal declarations with this
sequence leaves essentially no latitude, as might be expected. I believe the
required sequence to be as follows:

1. Process all declarations other than named-literal declarations, but
defer the translation of expressions in these declarations until
later (note that PRECISION attributes may contain expressions as a
consequence of extension #3).
2. Apply defaults to the above declarations to complete their attribute
sets.
3. Process named-literal declarations in order, including the evalua-
tion of their restricted expressions (the names of built-in
functions are now known).
4. Translate the expressions deferred from step 1. Previously, no
particular order was required for the translation of these expres-
sions. Now, however, some order is required. That is because the
translation of some of these expressions may involve generic
selection which can depend on a precision value, and yet precisions
may still be represented by untranslated and unevaluated restricted
expressions. Clearly these must be processed first.

Obviously, this must be subjected to fine scrutiny before one can be confident
of its correctness.
The erroneous dependence of one named literal on another declared later in the same block can be implicit, involving a DEFAULT statement as a link. For example, consider

```
DEFAULT (CONSTANT) PRECISION (M);
DECLARE N LITERAL (3);
DECLARE M LITERAL (4);
```

The declaration of N implicitly depends on M. For, during the translation of the restricted expression in the declaration of N, a constant to which the DEFAULT statement applies is encountered. (Defaults applicable to a constant take effect when the expression containing the constant is translated.) The constant 3 acquires PRECISION (M) by default, which is an error because M has not yet been declared. Had the default precision specification been PRECISION (N), the appearance of circularity would have been given; in reality, however, the error is identical.

Finally, a mundane matter. The attribute keyword LITERAL was chosen, instead of the expected CONSTANT, to avoid interference with the current uses of CONSTANT in the processing of defaults. (Recall that CONSTANT can be combined with FILE, ENTRY, FORMAT, or LABEL, and with DIMENSION, and that it does not take an operand.) The use of a new keyword greatly simplifies the task of integrating a new declaration type into the Standard.

(#5) Explicit precision specification for constants

If an arithmetic constant is not protected by the default suppression character, it can acquire a precision attribute by the default process. Only if no default supplies a precision does the constant then acquire the precision implied by its textual form. Standard language defaults specify precision attributes applicable to variables but not to constants; only user-written defaults can affect the precision of constants.

The capability described above is useful because it allows one to write floating-point constants with an excess of digits and have their precision attributes (and consequently the accuracy of their translated values) determined by default specifications that can be easily adapted to new environments; it is not necessary to rewrite any constants at the textual level. But this capability is not selective; it applies to all constants (of a given mode and scale) or to none. The same benefit, that of allowing a
constant to be written with excess precision and tailored conveniently to a lesser precision in a particular environment, cannot be extended to individual constants. In short, there is no way to specify an explicit precision for an individual constant independently of its textual form.

The required capability is provided by writing, as part of the constant, D (for "digits") followed by a parenthesized restricted expression, which is interpreted as an explicit number-of-digits in the base of the constant. The function of the default-suppression character, P, remains unaltered. It makes sense to restrict this explicit precision feature to floating-point constants (those that have an E, or that have neither an E nor an F and acquire FLOAT by default). Description and implementation costs are negligible.

(6) REQUIRE statement (a pragma)

This feature is intended to allow a program striving for portability to protect itself against certain adverse effects of overly restrictive quantitative limits that might be encountered in some implementations. An attempt to declare (or default) a precision in excess of the implementation maximum is already safe with regard to portability, because such an attempt is caught as an error. It is likewise an error to write a precision argument in a reference to an arithmetic built-in function which exceeds the implementation maximum. The same degree of safety does not extend, however, to the idea of a "derived precision", that is, a precision value multiplied or divided by 3.32 to reflect a change of base. In such instances the result of multiplying or dividing by 3.32 can exceed the maximum for the target base, and yet the Standard does not treat that as an error but merely limits the result to the maximum. Thus, in some implementations a particular conversion may be safe, while in others it could give rise during execution to an exceptional condition or, worse, inaccurate results. The numerical analyst, interested in accuracy, portability, and robustness, would prefer a compile-time diagnostic which implies that the implementation cannot meet his specifications to a subtle and sub rosa rewriting of those specs.

The facility proposed to assist in this area is the REQUIRE statement. Syntactically, the REQUIRE keyword is followed by a parenthesized restricted expression. The REQUIRE statement is a compile-time statement with utterly trivial semantics: the expression, evaluated with the attributes BIT (*),
must yield a bit result interpreted as "true", otherwise the program is in error. REQUIRE statements are translated and evaluated after declaration and default processing, including that of named literals. There are no order dependencies.

A solution to the above-described problem with derived precisions (in the sense of detecting the problem when it occurs) can be constructed easily with the REQUIRE statement. One need only devise an appropriate relational expression involving the declared precision of the source and the maximum precision of the target (obtained by reference to an environmental enquiry function).

The REQUIRE statement is a simple and general feature, amounting to a compile-time assertion. Although only one of its uses has been cited, others are easily imagined. Some are illustrated later.

3.2. Example

As a comprehensive example, a function procedure illustrating all six of these features is now presented. The example is definitely not a toy problem. In a literal sense it is, unfortunately, not realistic in that it is an implementation of a PL/I built-in function (namely, EXP); thus, no PL/I programmer would ever write this particular procedure. However, it is representative of the kind and quality of portable mathematical software produced by numerical analysts.

The procedure shown here is a further refinement and development of one contained in [8]. It is a realization in PL/I of an algorithm due to Cody and Waite [6]. Several versions of the algorithm are given in [6], each tailored to certain broad characteristics of the host machine. Specifically, there are separate versions for decimal and non-decimal (e.g., binary, octal, or hexadecimal) floating-point machines and for non-decimal fixed-point machines (those whose fixed-point operations are significantly faster than the floating-point operations). The algorithm as implemented here is the version for non-decimal floating-point machines. In each version, Cody and Waite use rational function approximations whose degrees depend on the desired accuracy; the implementation shown here uses an approximation good to about 29 bits of accuracy. It is general with regard to the hardware radix, which may be 2, 4,
8, or 16. It also has steps to counteract "wobbling precision" due to leading zeroes in the fraction when normalization is in units of more than one hit.

The program proceeds by first reducing the given argument, X, to a related argument, G, in a small interval symmetric about the origin (statements 35-37); then computing the exponential, R, of the reduced argument using rational function approximation (statements 38-43); and finally reconstructing the desired function value, Y, from its components (statements 44-47). Notes on each statement follow the procedure.

The treatment of exceptional conditions in this procedure requires some motivation. Obviously, values of the argument X can be received for which the exponential of X is not representable because it exceeds either the overflow threshold or the underflow threshold of the implementation. The appropriate response to such a situation in PL/I is to raise the OVERFLOW or UNDERFLOW condition and allow the user (caller) of the exponential procedure to trap it and do what he pleases. But how shall the condition be raised? In particular, can we guarantee that when X is "too large" some arithmetic operation in EXP will cause OVERFLOW, and that when X is "too small" some arithmetic operation will cause UNDERFLOW, and that no other condition (like SIZE or FIXEDOVERFLOW) can occur instead? The answer requires a deep analysis of the algorithm and its realization [8]. In most cases the answer is likely to be indeterminate because of the numerous contexts in which the Standard requires a value to be converted to "integer-type" with a precision determined by the implementation for each context. In the present example, therefore, I elect (as most numerical analysts would) to forego the required analysis and substitute instead tests on X that allow the computation to proceed only if X is "in range". If X is found to be "out of range", the appropriate condition is raised by signaling it in lieu of proceeding with the computation. The numerical analysis extensions allow these implementation-dependent tests to be made portable.

1. EXP: PROCEDURE (X) RETURNS (FLOAT);
2. DEFAULT (RANGE (*) | CONSTANT) BINARY;
3. DEFAULT (FLOAT & BINARY) PRECISION (WP);
4. DECLARE WP LITERAL (21);
5. DECLARE MAX X LITERAL (LOG(MAXFLT2(WP)));
6. DECLARE MIN X LITERAL (LOG(MINFLT2(WP)));
7. DECLARE SIZE X LITERAL (FLOOR(LOG2(MAX(MAX_X, -MIN_X)) + 1));
8. DECLARE HP LITERAL (WP + SIZE X + 1);
9. DECLARE LOGRADIX LITERAL (LOG2(RADIX2()));
10. DECLARE EPS LITERAL (EPSILON2(WP));
11. DECLARE LN2HP LITERAL (.6931471805599453094172321E0D(HP));
12. DECLARE LN2INV LITERAL (1.4426950408889634074E0);
13. DECLARE PI LITERAL (.41602886268E-2);
14. DECLARE PO LITERAL (.24999999950E0);
15. DECLARE Q1 LITERAL (.49987178778E-1);
16. DECLARE Q0 LITERAL (.5E0);
17. DECLARE (X, Y, Z, G, GP, Q, R) FLOAT;
18. DECLARE (N FIXED, XN FLOAT) PRECISION (SIZEX + 1);
19. DECLARE J FIXED PRECISION (2);
20. DECLARE POWERSOF TWO (3) FLOAT STATIC INITIAL (2E0, 4E0, 8E0);
21. DECLARE (MAXFLT2, MINFLT2, RADIX2, EPSILON2, SCALE, LOG, LOG2,
FLOOR, TRUNC, SIGN, SUBTRACT, MULTIPLY, MOD, ABS, MAX)
BUILTIN;
22. REQUIRE (WP <= 29);
23. REQUIRE (RADIX2() = 2 | RADIX2() = 4 | RADIX2() = 8 | RADIX2() = 16);
24. IF ABS(X) < EPS THEN Y = 1E0;
25. ELSE IF X > MAX X THEN DO;
26. SIGNAL OVERFLOW;
27. SIGNAL ERROR;
28. STOP;
29. END;
30. ELSE IF X < MIN X THEN DO;
31. SIGNAL UNDERFLOW;
32. Y = 0E0;
33. END;
34. ELSE DO;
35. XN = TRUNC(X*LN2INV + SIGN(X)*.5E0);
36. N = XN;
37. G = SUBTRACT(X, MULTIPLY(XN, LN2HP, HP), HP);
38. ON UNDERFLOW ;
39. Z = G*G;
40. GP = (PI*Z + PO)*G;
41. Q = Q1*Z + Q0;
42. REVERT UNDERFLOW;
43. R = .5E0 + GP/(Q - GP);
44. N = N + 1;
45. J = MOD(N, LOGRADIX);
46. IF J > 0 THEN R = R*POWERSOF TWO(J);
47. Y = SCALE(R, (N - J)/LOGRADIX);
48. END;
49. RETURN (Y);
50. END;

Notes on individual statements follow.

2. This DEFAULT statement supplies BINARY base for arithmetic variables declared without a base and for arithmetic constants.

3. This DEFAULT statement supplies a precision for FLOAT BINARY variables or constants that lack it. The base, BINARY, may have been supplied by
the previous DEFAULT statement. The PRECISION attribute contains a restricted expression consisting of a reference to the named literal, WP.

4. WP is declared as a named literal and is given a value by the restricted expression consisting of the constant 21. This constant will be translated to an "abstract constant" with BINARY base (a consequence of statement 2), FIXED scale, and converted precision 8. The attributes of WP are thus REAL FIXED BINARY PRECISION (8). The value of WP is to be supplied by the user of this subroutine before compilation. It is intended that the user assign to WP the value which represents the "working precision" of his computations involving EXP. For most of its computations and its variables, EXP itself will then use this same precision. All of the other specifications of EXP derive from WP and from environmental parameters accessed at compilation time by references to environmental enquiry functions. This realization of the algorithm for EXP will produce results accurate to approximately WP bits; no assumptions can be made about extra accuracy when the underlying hardware representation uses more than WP bits. (Note that declaration 4 is not affected by default 3 because the constant, 21, in statement 4 does not have the FLOAT attribute; if it did, an attempt to give it PRECISION (WP) would be detected as an error because during its translation WP does not have a value.)

5. A constant that depends on both WP and properties of the implementation is computed, during compilation, by this statement and given the name MAX_X. MAXFLT2(n) is an environmental enquiry function whose attributes are REAL FLOAT BINARY PRECISION (n) and whose value is the largest that the implementation can store in a variable declared with those attributes. (The analogous function for decimal base would be named MAXFLTIO.) The LOG of MAXFLT2(WP) is computed. Because LOG is generic, the attributes passed through to MAX_X are REAL FLOAT BINARY PRECISION (WP). Mathematically, MAX_X is approximately the largest value in the implementation that can be exponentiated without overflow (that is, the largest value whose exponential is representable). The approximation depends on the quality of the implementation's LOG function. Since the use of MAX_X is intended to avoid overflow in the computation of EXP(X), a value of MAX_X that is slightly "too large" is intolerable. An error of that nature really reflects more on the underlying implementation of PL/I than on the design of this program. Given, however,
that numerical analysts are forced to contend with the foibles of real computers and real compilers, the mathematical software they write usually exhibits certain defensive tactics necessary to achieve their goals of robustness. A fine line between prudence and paranoia must be trod. In the present example, however, I shall avoid such diversionary complications.

6. A similar constant, approximating the smallest (i.e., most negative) value in the implementation whose exponential is representable, is computed during compilation and named MIN\_X. The computation of MIN\_X is similar to that of MAX\_X, using the environmental enquiry function MINFLT2 instead of MAXFLT2. MINFLT2(n) has the attributes REAL FLOAT BINARY PRECISION (n) and a positive value which is the smallest that the implementation can store in a variable declared with those attributes. (The analogous function for decimal base would be named MINFLT10.) Additional properties which MINFLT2 (or MINFLT10) might need to satisfy, such as the representability of its reciprocal, depend on whose model of arithmetic one follows. Though this is a detail to be resolved eventually, it is not necessary to do so now. Finally, as before, in theory some defensive tactics are needed to guard against LOG returning a value which is slightly too small (too negative).

7. A constant whose value is interpreted as the number of bits required to represent faithfully the integral part of any X between MIN\_X and MAX\_X is computed and named SIZE\_X. The attributes of SIZE\_X work out to REAL FLOAT BINARY PRECISION (WP), although its value is an integer. Since that value is to be used only in computations leading to a precision specification, further coercion to fixed-point using FIXED might seem desirable. However, that is unattractive because the FIXED built-in function requires a second argument giving the precision of the converted result, and it seems pointless to worry about the precision of small integers (literally, the precision of a precision specification!). Once elaborated in detail, extensions to Standard PL/I that allow expressions in precision specifications will be seen to require converting the value of the expression from whatever attributes it has to "integer-type", which means that the precision of the converted value will be determined by the implementation; in this context, a number-of-digits sufficient to represent a value equal to the maximum precision that can be declared will do.
In the computation of \texttt{SIZE\_X}, \texttt{FLOOR} and \texttt{LOG2} are used to perform an operation equivalent to counting bits. This is both risky and inefficient. It is risky because typical implementations of \texttt{LOG2} may fail to give a precisely integral result when its argument is an integral power of two; if approximation or roundoff error produces a value slightly less than the correct integer, taking its \texttt{FLOOR} will yield a \texttt{SIZE\_X} that is too small by one whole unit. The precision specifications based on \texttt{SIZE\_X} will then be too small, and the \texttt{SIZE} condition could occur during execution. But avoiding conditions like \texttt{SIZE} is the whole point of computing portable precision specifications! One could "doctor" the expression with defensive adjustments, but that rapidly becomes unattractive. Since computation of integer values denoting precisions will be a common application of the numerical analysis extensions, an application-oriented built-in function to assist in a direct and error-free manner with that task is strongly suggested. For this purpose, \texttt{DIGITS2} would be useful. \texttt{DIGITS2}(x) would be defined as the least number of bits required for the integral part of \emph{x}. (\texttt{DIGITS10} measures the corresponding number of decimal digits.) \texttt{DIGITS2} and \texttt{DIGITS10} will return an integer-type result. The formula for \texttt{SIZE\_X} would then be, simply, \texttt{DIGITS2(MAX(MAX\_X, -MIN\_X))}.

8. HP means "higher precision". It is used to specify the precision of certain constants and operations involved in the computation of the reduced argument, \emph{G}, in statement 37. The formula for the value of HP is derived in the comments to that statement. As was the case for \texttt{SIZE\_X}, this named literal also has floating-point data type but an integral value. (However, if \texttt{DIGITS2} is used for \texttt{SIZE\_X} instead of \texttt{FLOOR} and \texttt{LOG2}, both would have integer-type.)

To appreciate the value of allowing named-literal declarations to refer to named literals declared earlier in the same block, the reader should pause here to consider how the restricted expressions in statements 5 through 8 would look without this provision. The one in statement 8 becomes two lines long and repeats many of the subexpressions of earlier statements, thus posing the hazard of a transcription error. In addition, the repetition of the user-selected constant "21" increases the user's difficulty of tailoring the procedure.
9. I compute here an auxiliary constant, LOGRADIX, which is a function of the radix of the implementation's hardware representation of REAL FLOAT BINARY variables. The environmental enquiry function RADIX2 provides the value of that radix. (To ascertain the hardware radix used for REAL FLOAT DECIMAL variables, RADIX10 would be used.) Since the hardware radix is an integer, the data types of RADIX2 and RADIX10 should probably be integer-type, as for other built-in functions (e.g., HBOUND) that return integral values. The formula is intended to yield the values 1, 2, 3, and 4 for hardware radices of 2, 4, 8, and 16, respectively. The discussion of the risks of using LOG2 and the inefficiency of using floating-point computations to map integers to integers, first presented in the notes to statement 7, is applicable here as well. If DIGITS2 is provided, the formula for LOGRADIX in terms of DIGITS2 is DIGITS2(RADIX2()) - 1.

10. Another auxiliary constant is here computed and named. This is the familiar "epsilon", i.e., the smallest positive floating-point number which when added to unity produces a result unequal to unity, in this case relative to the binary precision WP.

11-16. In the next several statements, simple floating-point constants written with excess precision are translated in conjunction with applicable defaults. In statements 12 through 16, DEFAULT statements 2 and 3 apply, so that the constants (and therefore the named literals) will have the attributes REAL FLOAT BINARY PRECISION (WP). The value of LN2INV in statement 12 is 1/LOG(2). The constant in statement 11, whose value is LOG(2), is given an explicit precision of HP (recall that the "higher precision" HP is derived from WP and environmental parameters). HP is interpreted as a binary precision because the constant is not protected by the default-suppression character, P, and thus acquires BINARY base from the action of DEFAULT statement 2.

17-20. The variables of the procedure EXP are declared. In statements 17 and 20, PRECISION (WP) is acquired by default. The precision SIZE*X + 1, explicitly declared in statement 18, is explained in the notes to statement

---

*This definition is informal. For further discussion, see [9].
35. The precision 2, explicitly declared in statement 19, is adequate because J's value is bounded by 3. Statement 20 declares a table of constant values used in statement 46, where it is involved in a subscripting operation. I am assuming, in consequence of the arguments in the section on named literals, that aggregates of named literals are not provided.

22. This REQUIRE statement guards against improper use of the procedure. It checks that the user has not attempted to tailor the procedure to values of WP in excess of those for which it was designed. In particular, the literals named in statements 11 through 15 have insufficient excess digits for working precisions higher than about 29; besides, for working precisions greater than 29 a rational function approximation of higher degree would be required. The Standard, supplemented with a REQUIRE statement, would say that the program is in error if WP is greater than 29, and a compiler would call attention to the violated requirement and reject the compilation at this point.

23. Similarly, this REQUIRE statement checks that the program is being compiled for a machine meeting the "non-decimal" requirement assumed for the algorithm. In fact, it checks for something even stronger: the hardware radix (for binary variables) must be 2, 4, 8, or 16. The program would be in error, and its compilation rejected, on any strictly decimal (base 10, even for binary variables) machine; on the TI home computer with Microsoft floating-point emulation, which uses a base of 100; and on the Russian ternary computer CETYHb. (I do not mean to imply that all these computers have PL/I!)

A REQUIRE statement limiting HP to values for which the constant in the declaration of LN2HP is adequately precise could be added.

24. This is the first executable statement. It is strictly an optimization, effecting a fast return in the likely case of a zero argument (and anything else sufficiently close to zero). Justification for the validity of the details of this statement follows from an error analysis of the exponential function and from the (informal) definition of "epsilon" (statement 10). From \( Y = e^x \) we obtain \( dY = e^x dx = Y dx \); therefore, the relative error in \( Y \), \( dY/Y \), equals the absolute error in \( X \), \( dx \). In the vicinity of the origin, from an absolute error in \( X \) of less than epsilon there results a relative error in \( Y \) of less than epsilon. But at \( Y = 1 \) the relative and absolute errors in \( Y \) are
the same. From the definition of epsilon, an absolute error less than epsilon at unity will not be observable.

Epsilon is used here in an atypical way. For a discussion of more typical uses, see [9].

25-29. Values of $X$ whose exponentials exceed the overflow threshold are excluded from further processing, and OVERFLOW (for which no ON-unit has been established in the current block) is signaled. Statements 27 and 28 protect against implementations which extend the Standard by allowing a return to the point of interrupt following the raising of OVERFLOW (or of ERROR), which is not allowed by the Standard even when the condition is signaled. The treatment of overflow (that is, abort unless an ON-unit provided by the user traps the condition and terminates with a GO TO statement) is harsh but identical to what would occur when any other operation raises an overflow.

30-33. The treatment of values of $X$ whose exponentials exceed the underflow threshold is similar but non-fatal. The invoking environment is given a chance to intercept a signaled UNDERFLOW condition (since no ON-unit has yet been established in the current block). Return from an UNDERFLOW ON-unit to the point of interrupt is allowed, and if it occurs, or if system action applies in the absence of an ON-unit, execution continues after returning a result of zero from EXP.

35. This statement begins the argument reduction for values of $X$ known to be in range. $X_N$ is computed as the nearest integer to $X/\log(2)$, i.e., at this point $|X_N - X/\log(2)| \leq 1/2$. Recall that SIZE $X$ was computed during compilation in statement 7 as the number of bits in the integral part of any in-range value of $X$. The precision required for $X_N$, as declared in statement 18, is one more than SIZE $X$ because $X_N$ is 44% larger than $X$.

An alternative computation of $X_N$ is provided by the formula $\text{SIGN}(X) \times \text{FLOOR}(\text{ABS}(X) \times \text{LN2INV} + .5E0)$. In either case, the Fortran definition of SIGN as the transfer of the sign of the second argument to the magnitude of the first is preferable. (Optimizing PL/I compilers can implement multiplication of a binary quantity by SIGN(...) as a sign transfer, and it is hoped that they actually do so.)
Incidentally, what guarantee have we that FLOOR or TRUNC or CEIL of a sufficiently small floating-point quantity yields the correct integer (as a floating-point quantity) with no approximation error? The Standard offers no such guarantee. It easily could, however, and certainly should. It already contains an obscure clause which requires that small integers yielded by the floating-point addition, subtraction, multiplication, or division of small integers be represented exactly; this clause should be broadened to encompass certain other operations, among them the three built-in functions named above, for which the guarantee of a precise small-integer result is desirable and implementable. Several problems of this nature exist; some are under study, but much more work is required. Additional discussion may be found in [9].

36. The value of XN is saved in a REAL FIXED BINARY PRECISION (SIZE\_X + 1) variable for later use.

37. The reduced argument is computed in "higher precision" and then saved in the working-precision variable G. G is the difference of two nearly equal quantities (X differs from XN\cdot\text{LOG}(2) by at most \text{LOG}(2)/2, which in binary is 0.0101100010111...). The precision, HP, in which the subtraction is performed must be high enough so that when the high-order digits cancel, as they will, at least WP or cancelling digits are left at the low-order end. The digits that cancel are the integral digits (of which there are up to SIZE\_X) and the first fractional digit. Therefore, HP needs to be SIZE\_X + 1 + WP or more. The worst case occurs when X is near either end of its valid range, when SIZE\_X + 1 digits will cancel, leaving WP. If X is, on the other hand, very small, fewer digits will cancel and more than WP will remain; on assignment to G, only the high-order WP of them will be retained. The built-in functions SUBTRACT and MULTIPLY are here employed in precisely the ways envisioned by the designers of PL/I. Slight improvement might be obtained if the right-hand side of statement 37 were rounded to WP bits, i.e., replaced by ROUND(SUBTRACT(...), WP).

At this point, \( X = N\cdot\text{LOG}(2) + G \), where \( |G| \leq \text{LOG}(2)/2 \). Therefore, \( e^{X} = e^{N\cdot\text{LOG}(2)+G} = 2^{N}e^{G} \).

38. If X is extremely close to a multiple of LOG(2), then G will be very close to zero. In this case an underflow could occur in statement 39, 40, or 41. If such an underflow occurs, it must not be reported to the caller (it
does not represent underflow in $e^X$). Consequently, a null UNDERFLOW ON-unit is established for the duration of these three statements only. On return from the null ON-unit back to the point of interrupt, the result of the operation which caused the underflow will be taken as zero.


42. With danger of underflow now passed, the establishment status of the UNDERFLOW condition is restored to what it was before statement 38 was executed. As an alternative to the REVERT statement, one could precede statement 38 by a BEGIN statement and change statement 42 to an END statement; this might be slightly less efficient than the REVERT.

43. The rational function approximation is here completed. At this point, $R$ approximates $e^G/2$ and lies between .5 and .707... (The factor of 1/2 counteracts "wobbling precision" in radices other than 2; that is, regardless of the radix the high-order bit of the normalized fraction is now 1.)

44. Because of the factor of 1/2 built into the value of $R$, $N$ is increased by one prior to (effectively) multiplying $R$ by $2^N$.

45-47. Computing $2^N$ is undesirable; depending on how it is performed, it can be expensive or imprecise. Toward finding a better way, note that multiplication by an arbitrary power of two is equivalent to multiplication by an appropriate power of the radix and then by a small residual power of two (if the radix is greater than two). For example, on a hexadecimal machine multiplication by $2^{27}$ is equivalent to multiplication by $16^6$ and then by $2^3$. Multiplication by a power of the radix, in turn, can be accomplished by simply incrementing or decrementing the exponent field by the desired amount. Because it is both fast and precise, an environmental manipulation function to scale in that manner by a power of the radix is desirable and has been proposed by numerical analysts. (On machines that lack a guard digit for multiplication, multiplying by a power of the radix -- even by unity -- can cause the loss of a whole radix-digit from the low-order end of the significand!) Such a manipulation function is of course very hardware-oriented, but its inclusion in a high-level language contributes to portability by eliminating from the programmer's concern all details about the location, width, and bias of the exponent field.
To see how the appropriate power of the radix and the small residual power of two are obtained from $N$, first note that $N$ can be expressed as $m \cdot \log_{\text{RADIX}} + J$, where $0 \leq J < \log_{\text{RADIX}}$. Multiplying by $2^N$ is then the same as multiplying by $2^{m \cdot \log_{\text{RADIX}} + J}$, i.e., by $2^{m \cdot \log_2(\text{RADIX}^2()) + J}$. This in turn is equivalent to multiplying by $\text{RADIX}^2()^{m \cdot 2^J}$ or, finally, to multiplying by $2^J$ and increasing the exponent by $m$.

Statement 45 computes $J$. Statement 46 increases $R$ by a factor of $2^J$ but omits the multiplication when $J = 0$. This is done not merely to achieve efficiency but also to avoid the unnecessary loss of a whole radix-digit on machines lacking a guard digit for multiplication.

Statement 47 scales $R$ by $m$ and delivers the result as $Y$. SCALE is the proposed environmental manipulation function for exact scaling by a power of the hardware radix (by incrementing or decrementing the exponent field). The attributes of SCALE are the attributes of its first argument, and the base attribute of that argument (rather than a suffix of 2 or 10 to the name) is used to specify whether scaling by a power of $\text{RADIX}^2()$ or of $\text{RADIX}^10()$ is to be performed. The expression for $m$ given in statement 47, i.e., the second argument of SCALE, specifies the amount by which the exponent of $R$ is incremented; it is a way of computing $\text{FLOOR}(N/\log_{\text{RADIX}})$ that avoids the complications of guaranteeing that $N/\log_{\text{RADIX}}$ yields fractional digits for $\text{FLOOR}$'s use (a non-obvious but messy problem).

3.3. Relationship to a preprocessor

Because the computation of expressions during compilation is suggestive of an activity typically performed by preprocessors, it seems relevant to evaluate the extensions proposed here in relation to the general capabilities of preprocessors.

It should be clear that all of the restricted expressions appearing in precision attributes and other places where precisions are specified, as well as those expressions (appearing in named-literal declarations) on which they depend, can in theory be evaluated in a preprocessor prior to the syntactic analysis of the source program proper. What is required of the preprocessor is an assignment capability and an adequate selection of data types (including floating point) and built-in functions (including the environmental enquiry functions). If one assumes a classical preprocessor whose overall effect is
to replace some textual fragments of the source program by other text, then the results of these calculations will be converted back to character form, ultimately, for transmission across the interface between the preprocessor and the compiler proper. Whether this is acceptable depends, perhaps, on the end-use of the computed quantities. In the example, for instance, some computations ended ultimately in integer values representing precisions, and these can be represented in character form without loss of accuracy. However, the final result of other computations was needed in floating-point form. In mathematical software the loss of accuracy due to extra conversions between floating-point and character form, and possibly the loss or suppression of inherited attribute information resulting from conversion to character, spells the difference between polished programs and mediocre ones. (Implementation of named literals as a "% REPLACE" feature exhibits the same drawbacks.) The integrated proposal, in contrast, affords the specialist no less control over the computation of "constant" floating-point quantities at compilation time than he can obtain during execution.

The success with which other aspects of the numerical analysis extensions can be subsumed by a text-replacement preprocessor also varies. The REQUIRE statement, for example, seems divorceable from the activities that take place during compilation proper; the facility for explicit precision of floating-point constants, on the other hand, is not. Thus, while some overlap exists between the numerical analysis extensions and the capabilities of a textual-replacement preprocessor, the latter cannot meet all the requirements of numerical analysts. The need for both exists.

It is easy to imagine further generalizations of some of these extensions, and it is tempting to try to push them far enough to accommodate the traditional applications and functions of a preprocessor. For example, little additional machinery would be needed to allow user-defined function-procedure references (with restricted expressions for arguments) in restricted expressions. A COMPILETIME attribute, perhaps as an alternative scope attribute in ENTRY declarations, may be all that is required to provide the compiler with the names, parameter-descriptors, and returns-descriptors of user-defined functions that it may invoke during the evaluation of restricted expressions. The procedures themselves will have been previously compiled and either linked with the compiler or made available for dynamic loading as
required. A similar mechanism, enriched by additional syntactical means of presenting arguments (like keyword-and-parenthesized-value), could possibly serve the more traditional text-replacement needs; there, the previously compiled procedures would have CHARACTER-valued parameters and results and would be invoked at a much earlier stage of the translation process (at about the time that % INCLUDE is handled). However, the ideas presented in this paragraph have not been rigorously researched and are not being seriously proposed at this time.

Acknowledgments

During the preparation of this report, William J. Cody and Brian T. Smith helped me acquire an appreciation for some of the more subtle problems facing numerical analysts, and Burton S. Garbow helped me to express my thoughts succinctly.

REFERENCES


