*THE RELATIONSHIP BETWEEN NUMERICAL COMPUTATION AND PROGRAMMING LANGUAGES, J.K. Reid (editor)*
*North-Holland Publishing Company*


# The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments

*W. Kahan and Jerome T. Coonen*

Mathematics Department
University of California
Berkeley, California 94720
U.S.A.

**We can improve numerical programming by recognizing that three aspects of the computing environment belong to intellectually separate compartments. One is the syntax of the language, be it Ada, C, Fortran or Pascal, which gives legitimacy to various expressions without completely specifying their meaning. Another might be called "arithmetic semantics". It concerns the diverse values produced by different computers for the same expression in a given language, including the values delivered after exceptions like over/underflow. The third compartment includes diagnostic aids, like error flags and messages; these too can be specified in language-independent ways. However imperfect, this decoupling should spell out for all concerned the nature of arithmetic responsibilities to be borne by hardware designers, by compiler writers and by operating system programmers.**

> "Another of the great advantages of using the axiomatic approach is that axioms offer a simple and flexible technique for leaving certain aspects of a language *undefined*, for example...accuracy of floating point... This is absolutely essential for standardization purposes..."
>
> – C. A. R. Hoare (1969)

Professor Hoare's attitude toward floating point semantics reflects the anarchy that befell commercial floating point hardware early in the 1960's [1], and worsened in the 70's. That anarchy confounded attempts to characterize all floating point arithmetics in one intellectually manageable way. Now there is hope for the 1980's. A new standard for binary floating point arithmetic has been proposed before the IEEE Computer Society, and a radix-independent sequel is in the works. Since the binary standard has been adopted by a broad range of computer manufacturers, including much of the microprocessor industry, we expect numerical programs to behave more nearly uniformly across different computers, and perhaps across different languages as well. A draft of the binary standard, along with several supporting papers, may be found in the March 1981 issue of *Computer* [2-5].

Starting in the 1960's programming language designers came to be the arbiters of most aspects of the programming environment. With control of the programmers' vocabulary, language designers could control fundamental features such as the number of numeric data types available and the extent of run time exception handling. The language even limited the numeric *values* available by constraining the literals in the source text. This is not to say that language designers acted capriciously. They were disinclined to mention any capability not available on all computers. In this respect computer architects have laid a heavy hand on the computing environment. Languages must reflect the least common denominator of available features, and so they tend to vague oversimplifications where floating point is concerned. An extreme case is the new language Ada which, by incorporating W. Stan Brown's very general model for floating point computation [6], pretends that the difference between one computer's arithmetic and another's is merely a matter of a few environmental parameters. But sometimes the

programmer must know his machine's arithmetic to the last detail, especially when trying to circumvent limitations in range or precision. These details, dangling between language designers and computer architects, too often receive short shrift from both. Tying up these loose ends would improve the computing environment.

Of course the computing environment invites numerous improvements, to graphics, file handling, database management and others, as well as floating point and languages. But enhancements to which high-level languages deny access are enhancements destined to die. Those of us working on the proposed IEEE floating point standards have had to face this problem. We believe the solution is a proper division of labor, rather than grand attempts to improve too many aspects of the computing environment simultaneously; the latter way would require impractical coordination. For example, to encourage independent development of programming languages and floating point hardware, we propose that language (syntactic) issues be decoupled from arithmetic (semantic) issues to the extent possible. We present our view of the interplay between syntax, semantics, and diagnostics as parts of the computing environment, and discuss how they interface with each other. Given an adequate interface discipline, we hope that responsibility for these parts can be divided among language designers, numerical analysts, systems programmers, and others. In the past this division has been unclear. Unfortunately, when everybody is responsible, or when nobody is responsible, then everybody can be irresponsible.

**Portability**

We regard the programming language as just one layer of the computing environment, dissenting from a more traditional view that the language is the environment. What does this mean for program portability? Until very recently, portability of numerical programs was considered to be a quality of source code that could be compiled and run successfully without change on a variety of computers. The issues appeared largely syntactic. For example, programs like the PFORT verifier [7] were developed to check Fortran codes for adherence to a standard for "portable Fortran", their principal task being to weed out various quirks of dialect. Nowadays, we acknowledge that the portability issues go deeper than differences among Fortran dialects. They entail the (semantic) subtleties of over/underflow and rounding that, if ignored, can cause ostensibly portable programs that function beautifully on one machine to fail on another. Programming languages that lack the vocabulary required to address these issues aren't very helpful here. If we cannot "mention" these issues how can we resolve them?

Ideally, the variation of floating point arithmetic from one machine to another should be describable with a few parameters [8] which portable programs could determine through system-dependent environmental inquiries [9]. This scheme works satisfactorily for many programs that do not depend critically upon the finer points of the arithmetic. However, any such parameterization must be based upon an abstract model encompassing simultaneously all current arithmetic engines, some of them disconcertingly anomalous [1, 10]. To insist that this model underlie portable programming is to dump upon programmers the onus to discover and defend against all mishaps the model permits, some of them mere artifacts of generality. This in turn would burden programs with copious tests against subtle (and certainly machine-dependent) thresholds to avoid problems with idiosyncratic rounding and over/underflow phenomena. A programmer who shirks his responsibility to produce robust code obliges the user of his program, possibly another programmer, to unravel a more tangled web. Ultimately, the buck may be passed to users who find either their programs or their computers to be inexplicably unreliable. We doubt that any semantic analog of the PFORT verifier will ever be able to test for robust independence of the underlying arithmetic. Computer arithmetics are too diverse to allow every potentially useful numerical algorithm to be programmed straightforwardly in a fashion formally independent of the underlying machine.



Portability at the source code level is nice when inexpensive. When not, we are content with "transportability", whereby algorithms can be moved from one environment to another by routine text conversion, possibly with some aid from automation. An algorithm may depend critically upon the underlying arithmetic semantics and upon a system's ability to communicate error reports between subprograms. It is transportable to the extent that the dependencies can be communicated in natural language using mathematical terms, if not in Fortran. We are not advocating yet another programming language. We prefer that programmers accompany their codes with some documentation that explains, and can even be used to verify, how the program handles its interactions with the underlying system. Because computing environments are so diverse, we expect some algorithms to be transportable to only a few systems, not all; this does not undermine the notion of transportability. Essential to transportability is a manageable corpus of information about

- syntax – the programming language to be used,
- semantics – the arithmetic of the underlying computer, including the run-time libraries of functions like cos(), and
- diagnostics – the system's facilities for error reporting and handling,

preferably no more than can fit on a short bookshelf, and yet enough to cover a wide range of manufacturers' equipments.

**Syntax**

In this paper, *syntax* refers to the expressions in a language – which ones are legitimate and how they are parsed. Issues relevant to numerical calculations include the number of data formats available, how they combine to form arrays and structures, and the order of evaluation in unparenthesized expressions. Languages vary greatly in their provision of numeric data formats, usually called "types". Both Basic and APL have just one numeric type, which is to be used for both integer and floating point calculations; Pascal and Algol 60 have just one real type. Fortran and C have single and double types, although in C all floating expressions are of type double. PL/I programmers may specify the precision of their floating point variables, though they typically map into the single and double types supported by the underlying system. The new language Ada provides syntactic "packages" in which floating types may be defined to correspond to the host system's facilities, but its strong typing prohibits mixing of different user-defined types in expressions without explicit coercions, even if the underlying hardware types are the same.

Expression evaluation is just as varied. For example, in

$$1.0 + 3/2$$

most compilers would recognize the 3 and 2 as integers. Their ratio would be evaluated as the real 1.5 or truncated integer 1 depending upon the strength of the 1.0 to coerce their types. Different Fortran compilers have disagreed in this situation. In Ada such an expression would be illegal unless the 3 and 2 were written with decimal points to indicate that they were real literals. What about the unparenthesized expression

$$A * B + C \quad ?$$

Most languages, like Fortran, evaluate it as if it were written $(A*B) + C$, but APL evaluates $A \times B + C$ as if it were written $A \times (B+C)$. The situation gets more complicated when relational and boolean operators are involved. In Pascal, the attempt to simplify the language by keeping the number of levels of operator precedence small led to some surprises for programmers. For example, because the conjunction $\cap$ has greater precedence than $<$, the expression

$$x < y \cap y < z \ ,$$

used for checking bounds on the variable $y$, has the bizarre interpretation

$$(x < (y \cap y)) < z$$

which is illegal because of the appearance of the real $y$ as an operand to $\cap$.

Perhaps the widest syntactic liberties are taken by standard C compilers. Expressions of the form

$$a + b + c \ ,$$

where $a$, $b$, and $c$ may be subexpressions, are evaluated in an order determined at compile time according to the complexity of $a$, $b$, and $c$. This is so *regardless of parentheses* such as

$$(a + b) + c \ .$$

Such a convention is disastrous in floating point where, say, $(a+b)$ cancels to a small residual to be added into the accumulation $c$. In such cases all accuracy may be lost if $(b+c)$ is evaluated first at the compiler's whim. The cautious programmer who writes

$$(x - 0.5) - 0.5$$

to defend against a machine's lack of a guard digit during subtraction will always be vulnerable, if not to a C compiler then to an optimizer that collapses the expression into the algebraically, though not numerically, equivalent form $(x - 1.0)$.

To jump the gun a bit, it is clear from the examples above that *syntax constrains semantics*. Syntax also constrains programmers who, C compilers notwithstanding, are well advised to preclude any ambiguity in expression evaluation by inserting parentheses liberally.

**Semantics**

We concentrate here on arithmetic semantics. That is, after an expression has been parsed – so the computer knows which operations to perform – what does its evaluation yield? Floating point semantics depends vitally on the underlying arithmetic engine. The initiated reader realizes that this is where the real headaches set in. For example, on machines such as programmable calculators where the fundamental constants $\pi$ and $e$ are available in a few strokes, we might expect

$$(\pi \times e) - (e \times \pi)$$

to evaluate to 0.0 since, *semantically*, we expect multiplication to be commutative despite roundoff. Unfortunately, even this simple statement is not universally true. Different Texas Instruments calculators yield different tiny values for the expression above; and it's not just a matter of machine size and economy, for early editions of the Cray-1 supercomputer exhibited similar noncommutativity.

Another well-known example of murky semantics is the expression

$$X - (1.0 \times X)$$

which is exactly $X$ rather than 0.0 for sufficiently tiny nonzero values $X$ on Cray and CDC computers. On these machines $(1.0 \times X)$ flushes to 0.0 for those tiny $X$. On some other machines that lacked a guard digit for multiplication, the expression above was nonzero whenever $X$'s last significant digit was odd!

Hardware-related anomalies like these seem to predominate in any serious treatment of arithmetic semantics. Such distractions are what led Professor Hoare to despair about floating point in high-level languages. We will not dig further into the lore of arithmetic anomalies. Interested readers can find an introduction in [1]. The technical report [10] studies the overall impact of anomalies and compares two approaches to improvement.



Arithmetic semantics is not restricted to simple operations. In languages like Basic that include matrix operations, assignments like

$$MAT\ X = INV(A) * B$$

are allowed. As users might expect, most implementations evaluate $(A^{-1}) * B$ (approximately), following the strict mathematical interpretation of the formula. However, more robust systems by Tektronix and Hewlett-Packard use Gaussian elimination to solve the linear system $AX = B$ for $X$, thereby obtaining a usually more accurate $X$ that is guaranteed to have a residual $B - AX$ small compared with $|B| + |A| \cdot |X|$. If $A$ is close enough to singular, the subexpression $INV(A)$ may be valid or not depending upon good or bad luck with rounding errors – on all machines except the Hewlett-Packard HP 85. All machines solve $(A + \Delta A)X = B$ with $\Delta A$ comparable to roundoff in $A$ though possibly differing from column to column of $X$. The HP 85 further constrains $\Delta A$ to guarantee that $(A + \Delta A)^{-1}$ exists. Thus it has no "SINGULAR MATRIX" diagnostic. Consequently, a program using inverse iteration to compute eigenvectors always succeeds on the HP 85 but on other machines is certain to fail for some innocuous data. Is such a program, using a standard technique, portable or not? Who is to blame if it is not?

Arithmetic exceptions such as over/underflow and division by zero fit into our informal notion of semantics when they are given "values". We take this view in spite of a current trend among authors to consider exceptions under a separate heading *pragmatics*. This trend is understandable, given the variety of exception handling schemes across different hardware. Consider for example the expression 0.0/0.0 . When they are to continue calculation (i.e. without a trap) CDC, DEC PDP/VAX-11, and proposed IEEE standard machines stuff a non-numeric error symbol in the destination field. This symbol is then propagated through further operations. Most other machines just stop, forcing program termination. At least one will store the "answer" 1.0.

Dividing zero by itself is usually bad news within a program, so the diversity of disasters that arise on various machines is not too surprising. A quite different situation arises with the exponentiation operator in $Y^X$. Since this is part of the syntax of several languages, for example Fortran, Basic, and Ada, responsibility for its semantics has been taken by language implementors. Of the many problems that arise we will consider just one: what is the domain of $Y^X$ when both $X$ and $Y$ are real variables? Consider the simple case $(-3.0)^{3.0}$, which is:

| | |
|---|---|
| −27.0 | ...on very good machines, |
| −26.999...9 | ...on good machines, |
| *TERMINATION* | ...on bad machines, |
| undefined | ...on cop-outs, |
| +27.0 | ...on *very* bad machines. |

Why this bizarre diversity of semantics? Although for arbitrary $X$ the expression $Y^X$ may have no real value when $Y$ is negative, the particular case above is benign because $X$ has an integer value 3.0. Thus restricting the domain of $Y$ to nonnegative numbers is unnecessarily punitive. We recommend that, should $X$ be a floating point Fortran variable with a nonzero integer value,

$$Y ** X = Y ** \mathrm{INT}(X) \ .$$

This cannot hurt Fortran users, but will help the Basic programmer (and the conversion of programs from Basic) because most implementations of Basic, with just one numeric data type, cannot distinguish the real 3.0 from the integer 3 in the exponent. This recommendation costs extra only when $Y$ is negative. On the other hand, if $Y$ is 0.0 we distinguish $Y^{0.0}$, which is an error, from $Y^0 = 1.0$ which mathematics makes obligatory. Note that none of these issues are language issues, though until now they have been settled by language implementors. Ideally, these responsibilities should be lifted from language designers and implementors, and

borne by people like the members of IFIP Working Group 2.5.

The point of this digression into the murk of pragmatics was to indicate that the current situation in exception handling is the result of a host of design flaws rather than inherent difficulties. We object to the connotation "pragmatics" carries with it of acquiescence to inevitable hazards. We prefer to capture all semantics, including the anomalies, under one heading even if this entails a different semantics for each different implementation of arithmetic. This exposes rather than compounds a bad situation.

A notably clean and complete arithmetic semantics is provided by the proposed binary floating point standard. The IEEE subcommittee responsible for the proposal set out to specify the result of *every* operation, balancing safety against utility when execution must continue after an exception. Even a cursory glance at the proposal indicates the extent to which exception handling motivated the design:

- Signed $\infty$ for overflow and division by 0.0.
- Signed 0.0 to interact with $\pm\infty$, e.g. $+1.0/-0.0 = -\infty$.
- NaN – not a number – symbols for invalid results like 0.0/0.0 and $\sqrt{-3}$.
- Denormalized numbers – unnormalized and with the format's minimum exponent – to better approximate underflowed values.
- Sticky flags for all exceptions.
- Optional user traps for alternative exception handling.

These features promote comprehensible semantics for "standard" programming systems.

### Diagnostics

After syntax and semantics, the third aspect of the numerical programming environment is the set of execution time diagnostic aids. They may be roughly divided into anticipatory and retrospective aids, and according to whether they find use during debugging or during (robust) production use.

The principal anticipatory debugging aid is the breakpoint for control flow and, when the hardware permits, for data too. Some systems can monitor control or data flow according to compiler directives inserted in a program. Retrospective debugging aids include the familiar warnings and termination eulogies, as well as the more voluminous memory dumps and control tracebacks. Systems with sticky error flags can list those still standing when execution stops – in a sense they signal unrequited events.

For the production program that would be robust, and perhaps even portable, the situation is not so clear. Because most current systems provide neither exception flags (such side effects are anathema to some language designers) nor error recovery, a program – if it is not to stop ignominiously on unusual data – must include precautionary tests to avoid zero denominators and negative radicands, and tests against tiny, but carefully chosen, thresholds to ward off the effects of underflow to zero. The lack of flags can force the use of explicit error indicators in subprogram argument lists to communicate exception conditions. The languages Basic, PL/I, and Ada allow for anticipatory exception handlers (e.g. ON <condition> ... in PL/I) but do not allow the exception handler to discover anything about the exception beyond a rough category into which it has been lumped, thereby making an automatic response by the program very cumbersome.

Another variety of anticipatory diagnostic aid is available through an option in the proposed floating point standard. It is essentially an extension of the PL/I "on-condition" except that it is outside any current language syntax. This feature, which might be called trap-with-menu, allows the programmer to preselect from a small list of responses an alternative to the default response. By devising the menu carefully, we should be able to give the user sufficient flexibility without having to cope with a voluminous floating point "state" at the time of the exception.



### The Syntactic–Semantic Interface

From the point of view of the numerical analyst, the semantic content of programming languages is given by the following list.

- What are the numeric types, and what is their range and precision?
- Which numeric types are assigned to anonymous variables like intermediate expressions, converted literals, arguments passed by value,...?
- Which numeric literals are allowed, and are they interpreted differently in the source code than the IO stream?
- Which basic arithmetic operations are available, and what is in the library of scientific functions?
- Is there a well-understood vocabulary reserved for the concepts and functions we need, and defended against collision with user-defined names?
- What happens when exceptions arise? How can error reports be communicated between subprograms?
- Is there a way to alter the default options (for, say, rounding or handling of underflow) by means of global flags?

These are among the knottiest issues in numerical computation. But, to a large extent, they can be freed from the more conventional language issues and thus resolved within the numerical community. Only questions about data types and the change of control flow on exceptions are necessarily tied to language syntax.

Consider a hypothetical language with only skeletal numerical features. Assume that integer types and arithmetic and character strings are "fully" supported. The language supports single and double real variables, pointers to them, and allows real variables to be embedded in arrays and structures. There is also provision for functions returning real values, and for real parameters passed either by value or reference. But the *only* operation on real types is assignment of a single value to a single variable, and of a double value to a double variable.

To be useful numerically, this hypothetical language would require a support library providing the basic arithmetic operations as well as the usual complement of elementary functions. But because each operation more complicated than a straight copying of bits would result only from an explicit function call, the programmer would in principle have complete control of the arithmetic semantics (by choosing a suitable library). As an example, consider the evaluation of the inner product of the single arrays $x[\,]$ and $y[\,]$ using a double variable for the intermediate accumulation to minimize roundoff:

```
double_precision temp_sum;
temp_sum := DOUBLE_LITERAL( "0.0" );
for i in 1 .n do
      temp_sum := DOUBLE_SUM( temp_sum,
            SINGLE_TO_DOUBLE_PRODUCT( x[i], y[i]) ); od
inner_product := DOUBLE_TO_SINGLE( temp_sum );
```

Even this simple example exposes many of the questions that arise in numerical programs. Would the constant 0.0 require a special notation (such as 0.0D0) to be assigned to a double variable? In a more conventional rendition of the program the inner loop would involve a statement of the form

```
temp_sum := temp_sum + x[i]*y[i];
```

Would the product be rounded to single precision before the accumulation into *temp_sum*, destroying the advantage of double precision?

**Semantic Packages**

The skeleton language above may be unambiguous, but it is clearly much too cumbersome for calculations involving complicated expressions. What we must do is bridge the gap between the handy syntactic expression $x[i]*y[i]$ and the semantically well-defined

SINGLE_TO_DOUBLE_PRODUCT( x[i], y[i]) .

We propose to do this through so-called semantic packages.

It may be a sign of progress that the new language Ada comes very close to suiting our needs. Although Ada incorporates the Brown model for arithmetic by providing a set of predefined attributes for each real type available to the programmer, this is in general insufficient for programs that would be robust. More important for us, Ada allows the overloading and redefinition of the infix operators +, −, etc. and in so doing provides the *explicit* connection between the operators and the real hardware functions they represent. The semantic packages, corresponding directly to the (syntactic) **packages** construct in Ada, could contain exact specifications of the arithmetic functions (which are actually implemented in hardware). Thus there would be a semantic package for each basic architecture, for example IBM 370, DEC PDP/VAX-11, and the proposed IEEE binary standard. Some semantic packages could be more general, encompassing several machines whose arithmetic is similar enough that a few environmental inquiries supply all the distinction that is necessary for a wide range of applications. For example, one such package might include IBM 370, Amdahl, Data General MV/8000, HP 3000, DEC PDP/VAX-11 and PDP-10, relegating TI, CDC 6000, Cray 1 to another.

Our attempt to force the gritty details of arithmetic semantics upon programmers may dismay readers who embrace the modern trend to elevate the programming environment above machine details. Such an attempt is made within Ada, by means of a small set of predefined attributes associated with each real type. We have already explained that this is not enough; sometimes the program that would be robust must respond to machine peculiarities that defy simple parameterization. The report [10] on why we need a standard contains several examples.

An effort to "package" arithmetic semantics within various programming languages may seem impossible. For example, the details of floating point, especially in the proposed IEEE standards, involve global flags to indicate errors, and modes to determine how arithmetic be done. In Fortran, such state variables may be defined as local data within the standard library functions whose job is to test and alter the flags, although the actual implementation involves collusion with the hardware flags. This is not a complete formalization, since Fortran provides no way to describe the connection between the flags and the arithmetic operations. Current trends in language design eschew error flags as side effects of the arithmetic operations (functions). Modes and flags seem to violate the principle that all causes and effects of expression evaluation should be visible within that expression. Perhaps surprisingly, Ada again provides us with the desired facility – but without excessive or expensive generality. In accordance with the Steelman requirements of the United States Department of Defense, Ada permits side effects " limited to own variables of encapsulations". This is exactly our intention in using semantic packages to describe arithmetic.

**Optimization**

Any treatment of floating point semantics must deal with that favorite whipping boy, the code optimizer. We considered a most extreme example above, in which C compilers would calculate floating sums like

$$(a + b) + c \ ,$$

without regard to the parentheses, in whatever order makes best use of the register file. This is simply a mistake in the language design



Not all anomalies are so clear-cut. Some questions arise when, as in architectures suggested by the proposed IEEE standard, extended registers with extra precision and range beyond both single and double types are used as intermediate accumulators. Consider the typical code sequence

```
x := a * b;
y := x / c;
```

in which all variables are assumed to be of type single. If $(a*b)$ were computed in an extended register, should that value or the single value $x$ be used in the evaluation of $y$? Efficiency dictates the former, saving one register load and lessening the risk of spurious over/underflow. But common sense dictates the latter, so that what the programmer sees is what the programmer gets.

A similar situation arises in inner product calculations of the type discussed above. Consider the loop

```
double_precision temp_sum;
temp_sum := 0.0;
for i in 1..n do
        temp_sum := temp_sum + x[i]*y[i]; od
inner_product := temp_sum;
```

in which, like the earlier example, all variables are single except for the double *temp_sum*. The fully "optimized" compiler might run this loop with just two extended registers, one to compute the products $x[i]*y[i]$ and one to accumulate *temp_sum*, thereby avoiding $(n-1)$ register loads and stores by simply keeping *temp_sum* in a register. Alas, the programmer asked for a double precision intermediate, not extended, so such optimization is precluded.

The moral of these examples is that declared types must be honored. Also, the type assigned by the compiler to anonymous variables must be deducible syntactically, or, better, it should be under the programmer's control. The alleged optimizations above were disparaged because named variables were replaced surreptitiously by extended counterparts that happened to be in registers. This is not to say that extended evaluation is unhealthy; on the contrary, extended temporaries can reduce the risk of spurious over/underflow or serious rounding errors, and therefore should be used for anonymous variables. But the advantage of extended is lost if languages prevent programmers from requesting it for declared temporaries. The expression

$$temp\_sum + x[i]*y[i]$$

in the loop above would best be computed entirely in extended before the store into *temp_sum*. These facilities for extended expression evaluation are not unique to the proposed IEEE standard; the benefits of wide accumulation were realized in the earliest days of computing. The Fortran 77 standard includes some intentionally vague language about expression evaluation in order not to prohibit extended intermediates, and the Ada standard, which seems to avoid some problems by strict typing and requirements for explicit type conversions in programs, uses a so-called *universal_real* type (at least as wide as all supported real types) for the evaluation of literal expressions at compile time.

The use of an extended type for anonymous variables is prone to one class of problems. When real values or expressions may be passed by value to subprograms there may be a conflict between the implicit type of the expression and the declared type of the target formal parameter. This problem arises in current implementations of the language C, which supports both single and double types but specifies that all real expressions are of type double. Suppose that a C program contains the statement

$y := f(a*b/c);$

where all variables are of type float (single) and the function $f()$ is defined by

```
float f(x)
  float x;
{ ..... }
```

How can the type of the expression $(a*b/c)$ be double while the type of the formal parameter $x$ is float? C resolves the discrepancy by silently countermanding the declaration of $x$ and replacing **float** by **double**. Once again, what you see is not what you get. This use of wider intermediates, exploiting the PDP-11 floating point architecture, is exactly analogous to one use of extended registers. Though it is efficient and straightforward to implement, it is not acceptable.

## Conclusion

We have cited examples to show that progress in numerical computing has been slowed by questionable decisions in the design of computing languages and systems. We have suggested a rough division into three categories, syntax, semantics and diagnostics, so that the difficult issues could be resolved by those most qualified – and most profoundly impacted. IFIP Working Group 2.5 might well take responsibility for the interfaces with semantics. Ideally their efforts will lead to fully specified environments for which reliable numerical software can be derived, possibly automatically, from algorithms expressed in a mathematical form if not already in a programming language. Programming then becomes a three phase translation involving the language (syntax) to be used, the underlying arithmetic engine (semantics), and the host system (diagnostics). We acknowledge that these categories are not completely independent, and that the boundaries between them cannot be drawn precisely, at least not yet. Nonetheless, we remain convinced that those boundaries must be drawn if we are to bring the required expertise to bear on the current morass.

## Acknowledgement

This report was developed and originally typeset on a computer system funded by the U. S. Department of Energy, Contract DE-AM03-76SF00034, Project Agreement DE-AS03-79ER10358. The authors also acknowledge the financial support of the Office of Naval Research, Contract N00014-76-C-0013.

## References

[1] Kahan, W., "A Survey of Error Analysis," in: *Information Processing 71*, (North-Holland, Amsterdam, 1972) 1214-1239.

[2] "A Proposed Standard for Binary Floating-Point Arithmetic," Draft 8.0 of IEEE Task P754, with an introduction by D. Stevenson, *Computer*, 14, no. 3, March (1981) 51-62.

[3] Cody, W. J., "Analysis of Proposals for the Floating-Point Standard," *Computer*, 14, no. 3, March (1981) 63-68.

[4] Hough, David, "Applications of the Proposed IEEE 754 Standard to Floating-Point Arithmetic," *Computer*, 14, no. 3, March (1981) 70-74.

[5] Coonen, Jerome T., "Underflow and the Denormalized Numbers," *Computer*, 14, no. 3, March (1981) 75-87.

[6] Brown, W. S., "A Simple But Realistic Model of Floating-Point Computation," to appear in *ACM Transactions on Mathematical Software*, 1981.

[7] Ryder, B. G., "The PFORT Verifier", *Software – Practice and Experience*, 4 (1974) 359-377.

[8] Sterbenz, P. H., *Floating-Point Computation* (Prentice-Hall, Englewood Cliffs, N.J., 1974).



[9] Brown, W. S. and S. I. Feldman, "Environment Parameters and Basic Functions for Floating-Point Computation," *ACM Transactions on Mathematical Software*, 6 (1980) 510-523.

[10] Kahan, W., "Why do we need a standard for floating point arithmetic?", Technical Report, University of California, Berkeley, CA, 94720, February (1981).