AD-A050 657 INSTITUTE FOR DEFENSE ANALYSES ARLINGTON VA SCIENCE A--ETC F/G 9/2
RATIONALE FOR FIXED-POINT AND FLOATING-POINT COMPUTATIONAL REQU--ETC(U)
JAN 78 D A FISHER, P R WETHERALL DAHC15-73-C-0200
UNCLASSIFIED P-1305 IDA/HQ-77-19836 NL

| OF |

AD A050657

END
DATE
FILMED
4 -78
DDC

IDA PAPER P-1305

RATIONALE FOR FIXED-POINT AND FLOATING-POINT COMPUTATIONAL REQUIREMENTS FOR A COMMON PROGRAMMING LANGUAGE

David A. Fisher
Philip R. Wetherall

January 1978

Prepared for
Defense Advanced Research Projects Agency

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

The work reported in this document was conducted under contract DAHC15 73 C 0200 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER 14 IDA Paper P-1305 ✓	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 Rationale for Fixed-Point and Floating-Point Computational Requirements for a Common Programming Language.		5. TYPE OF REPORT & PERIOD COVERED 9 Final rept. Jan [redacted]75-Jan [redacted]77,
		6. PERFORMING ORG. REPORT NUMBER 14 P-1305
7. AUTHOR(s) 10 David A/ Fisher, Philip R/ Wetherall		8. CONTRACT OR GRANT NUMBER(s) 15 DAHC15-73-C-0200
9. PERFORMING ORGANIZATION NAME AND ADDRESS INSTITUTE FOR DEFENSE ANALYSES 400 Army-Navy Drive Arlington, Virginia 22202		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ~~DARPA Assignment 37~~
11. CONTROLLING OFFICE NAME AND ADDRESS DEFENSE ADVANCED RESEARCH PROJECTS AGENCY 1400 Wilson Boulevard Arlington, Virginia 22209		12. REPORT DATE 11 Jan[redacted]978
		13. NUMBER OF PAGES 68 12 70p.
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) 18 IDA/HQ, SBIE 19 77-19836, AD-E500 019		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION DOWNGRADING SCHEDULE N/A

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Fixed Point, Floating Point, Numeric Computation, Scale, Range, Precision, Programming Languages, Number Representations, Common Language, Numeric Compilation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This paper discusses the considerations that led to the individual technical requirements for the numeric computation facilities for a common programming language for the Department of Defense (DoD). Of five kinds of arithmetic considered, one floating-point and one form of fixed-point including integers were found to be appropriate for the common language. The implications of the various requirements for the language designers, the compiler-writer, the user, and the machine

→ next page

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

403108

20.

cont. designer are considered.

IDA PAPER P-1305

RATIONALE FOR FIXED-POINT AND FLOATING-POINT COMPUTATIONAL REQUIREMENTS FOR A COMMON PROGRAMMING LANGUAGE

David A. Fisher
Philip R. Wetherall

January 1978

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
400 Army-Navy Drive, Arlington, Virginia 22202

Contract DAHC15 73 C 0200
ARPA Assignment A-37

ABSTRACT

This paper discusses the considerations that led to the individual technical requirements for the numeric computation facilities for a common programming language for the Department of Defense (DoD). Of five kinds of arithmetic considered, one floating-point and one form of fixed-point including integers were found to be appropriate for the common language. The implications of the various requirements for the language designers, the compiler-writer, the user, and the machine designer are considered.

CONTENTS

TABLES

I. INTRODUCTION

A. PURPOSE

The technical requirements for a common programming language have been developed through a long and tedious process involving many considerations. For the most part, those considerations and the rationale for individual requirements have not been documented adequately. The requirements are detailed in the *Revised IRONMAN* [1]. An older version of the requirements, the *WOODENMAN* [2], provided a brief rationale for many of the technical requirements. The background to the project and general design criteria for the language were discussed in a paper presented at the 1977 Computers in Aerospace Conference [3].

This paper consolidates and expands the previous documentation to provide a more complete understanding of the depth, breadth, and soundness of the fixed-point and floating-point requirements. It is an attempt to (a) document some of the considerations that led to the fixed-point and floating-point requirements, (b) show the strengths and weaknesses of the arguments supporting each requirement, and (c) point out both their desirable and undesirable implications.

B. AUDIENCE

The intended audience for this paper comprises four groups: potential users of the common language, language designers, compiler writers, and hardware designers. When appropriate, the discussion is aimed specifically at one or another of the groups. The considerations that led to the requirements involve all four groups, but above all, the requirements are supposed to reflect the needs of potential users. These needs are not only for certain capabilities in the language, but also for a language design that will aid the production of reliable and maintainable programs, that can become a common language, and that is practical to use. The problems of implementing compilers and optimizing object code have also influenced the requirements. Finally, many of the difficulties that users have to contend with in the arithmetic of existing programming languages are a direct reflection of hardware design errors that the language designer has passed on to the user.

C. SCOPE

This paper is restricted to discussion of the technical requirements that affect the numeric processing facilities of a common programming language for the Department of Defense (DoD). The relevant requirements are reproduced as the Appendix.

D. CAUTION

This report is provided only as an aid to understanding the rationale that led to specific technical requirements. Information presented here represents the opinions of the authors and in no way alters the meaning of the technical requirements presented in the *Revised IRONMAN.*

E. ORGANIZATION

The remainder of this chapter is divided into four sections. Section F provides a brief history of the common language effort, Section G discusses the methodology used in developing the technical requirements, Section H gives the overall philosophy of the requirements, and Section I lists the major assumptions that influenced the requirements for the fixed-point and floating-point facilities. Chapter II explains the five kinds of arithmetic that were considered. Chapter III discusses the requirements for a fixed-point facility in the form of scaled integers.

Chapter IV discusses the requirements for a floating-point facility.

F. HISTORY OF THE COMMON LANGUAGE EFFORT

The common language effort began in 1974 when groups in each of the Military Departments independently proposed the adoption of a common language for developing major defense systems. In January 1975, a joint service program was formulated on the advice of the Director of Defense Research and Engineering (DDR&E*) [4]. Activities of this program, the common programming language effort, are coordinated by the High Order Language Working Group (HOLWG) which is composed of official representatives from the Army, Navy, Air Force, and Marine Corps, and from the Defense Communications Agency (DCA), the National Security Agency (NSA), and the Defense Advanced Research Projects Agency (DARPA). The HOLWG is chaired by a representative of USD(R&E). Individuals from the National Aeronautics and Space Administration (NASA), the Office of the Assistant Secretary of Defense for Manpower, Reserve Affairs, and Logistics (OASD(MRA&L))and OASD-Comptroller also participate. The authors of this paper act, respectively, as the technical advisor to the HOLWG and as the representative of the British Ministry of Defence to the HOLWG.

The major concerns of the common language effort are to reduce the number of programming languages used in the DoD, and to develop facilities to control, distribute, support, and provide training for those that remain. Early in the effort, it was realized that it would be impractical to convert existing programs to a common language. Thus, the common language is intended only for new software systems and should not affect existing programs. The intended applications are restricted to

*Now Undersecretary of Defense for Research and Engineering (USD(R&E)).

embedded computer systems because they represent the majority of costs, and, unlike data processing and scientific applications, do not currently benefit from a common language.

G. DETERMINING THE TECHNICAL CHARACTERISTICS

The HOLWG was chartered to formulate the requirements for common high-order languages for DoD. In the spring of 1975, the HOLWG began an effort to determine the characteristics of a general-purpose programming language that would be suitable as a common language for embedded computer applications of the DoD. The characteristics were to be in the form of requirements which would act as constraints on the acceptability of a language, but would not dictate specific language features.

While there are several widely accepted general goals and criteria (such as efficiency, reliability, readability, simplicity, implementability), they do not lend themselves to quantifiable assessment. At the opposite extreme are specific language features advocated by some, which if adopted as requirements, would impose strong constraints on the form, but not necessarily increase the effectiveness of the language. The arguments for and against any specific language feature are applicable to a class of features sharing certain properties, and often depend on the other characteristics of the language. The requirements attempt to isolate the needed properties from the features that implement those properties. Initially, rigorous definition at the level of requirements proved difficult as a position to be confuted, so a *STRAWMAN* of preliminary requirements was established. The *STRAWMAN* was widely circulated within the Military Departments and, to a lesser extent, in the academic community and industry.

The reviews of the *STRAWMAN* resulted in inputs which were formed into a fairly complete, but still tentative, set of requirements called *WOODENMAN*. This document contained descriptions of the general (i.e., nonquantifiable) characteristics which were desired, together with many desirable characteristics whose feasibility, practicality, and mutual compatibility had not been tested. The *WOODENMAN*, too, was widely distributed, not only within the Military Departments but also to other government agencies, to the computer science research community, and to industry. Additionally, a number of technical experts from outside the United States were solicited for comments, the European community being especially responsive.

Based on the various inputs and the official responses from each of the Military Departments, a *TINMAN* set of requirements was derived [5&6]. The *TINMAN* removed (former) requirements for which there was no sound rationale, restricted requirements that were unnecessarily general, and modified others to be practical within existing technology. Each requirement in the *TINMAN* had its own justification which was outlined in the document and each individual requirement was judged to be feasible. The *TINMAN* requirements were officially approved by the Assistant Secretary for Research and Development of each of the Military Departments in January 1976.

Wide distribution of the *TINMAN* (to contributors and other interested groups) followed, and for a year comments were received. In addition, a workshop [7] was held at Cornell University in October 1976 to involve DoD representatives and the programming language research community in joint discussions of technical issues that had been raised regarding the requirements and to further investigate their feasibility.

Also, during 1976, 23 programming languages were evaluated against the *TINMAN*. These evaluations were performed by 16 companies and contractors. Most of the languages received at least two evaluations with, in most cases, the designers of a language included among its evaluators. The consensus of the evaluators was that it is currently possible to produce a single language that will meet the requirements [8]. That is, no technological impediment to a single language was found, and it is likely that potentially conflicting requirements, such as those for readable programs, avoidance of unnecessary complexity, implementable compilers, semantic and syntactic consistency, machine independence, and object code efficiency, can be met.

The languages that were evaluated included some currently being used for embedded computer applications in the DoD (e.g., CMS-2, JOVIAL, SPL-1, and TACPOL), languages used for process control and similar applications in Europe (e.g., CORAL-66, LIS, LTR, PEARL, and RTL/2), research languages that were known to satisfy specific requirements (e.g., EUCLID, MORAL, and ECL), and languages that are widely used outside the DoD (e.g., COBOL, FORTRAN, PASCAL, and PL/1). As might be expected, the more modern languages tended to satisfy the requirements for reliability and simplicity, while the languages for process control and DoD applications more nearly satisfied the requirements that reflect the special needs of embedded computer applications.

All of these efforts contributed to a new version of the requirements, called *IRONMAN*, that was issued in January 1977 [9]. The *IRONMAN* requirements are substantially the same as those of the *TINMAN*, but have been modified for feasibility and clarity, and are presented in a different format. The *TINMAN* was discursive and organized around general areas of discussion. The *IRONMAN*, on the other hand, is very brief and organized like a language description or manual. The *IRONMAN* is still sufficiently general to constrain the structure of a language without dictating the details of its design.

The most recent revision, the *Revised IRONMAN*, was issued in July 1977 and is available for comment [1]. This revision incorporates four kinds of changes. Most are clarifications, some remove redundancies that became apparent with the revised format, a few correct errors and inconsistencies that remained from previous revisions, and the remainder deal with special cases that can arise from interactions between the features of a language.

Each iteration of the requirements, beginning with *WOODENMAN,* has reduced the number and generality of the capabilities requested. The requirements have become more precise, although less constraining, as applications have become better understood, as application software has been examined with respect to known language features, and as more emphasis has been placed on the general requirements for reliability, maintainability, and efficiency.

At each iteration, the tentative requirements were distributed by the HOLWG and comments and suggestions were collected and coordinated by the Services. The resulting inputs were analyzed and integrated into a consistent set of requirements by the Institute for Defense Analyses (IDA).

One surprising result of the requirements effort has been the similarity of the requirements within the different application areas. Early in this program, there was a feeling that different user communities might have fundamentally different requirements with insufficient overlap to justify a common language or might have critical requirements that were incompatible. Such communities include avionics, guidance, command and control, communications, and training simulators. It has been impossible to single out different sets of requirements for particular communities. Almost all the potential users have the same requirements at the level of language characteristics, but different priorities. Often the priorities vary among segments of a given task. All users need input-output, real-time facilities, strong data typing, etc. Upon reflection, the technical rationale for this is clear. The surprise was historical and based on the observation that in the past the different communities have favored different language approaches. Further investigation showed that the origin of this disparity was primarily administrative rather than technical. This did not, however, establish that a single language could meet all the stated requirements, only that, if a language meeting all requirements were found, it would satisfy the perceived needs.

In all 74 commands and offices within the DoD, 66 individuals outside of DoD, and 43 companies and organizations (not counting the workshop at Cornell or the language evaluation efforts) have contributed over 2000 written pages of commentary on the requirements. Not all of their suggestions have been adopted and many have been modified before acceptance, but each has been considered in sufficient detail to determine whether it should or should not be followed.

H. PHILOSOPHY OF THE TECHNICAL REQUIREMENTS

The technical requirements reflect six major goals for the common language: (a) that it be suitable for software in DoD embedded computer systems; (b) that it be appropriate for the design, development, and maintenance of reliable software for systems that are large, long-lived, and continually undergoing change; (c) that it be suitable as a common language (i.e., complete, rigid, and machine-independent standards can be established); (d) that it will not impose execution costs in applications where it provides unused or unneeded generality; (e) that it provide a base around which a useful software design, development, maintenance, and support environment can be built; and (f) that it be an example of current good language design practice. At the highest level, the technical requirements take the form of general design criteria that are most strongly influenced by the first three goals above.

The characteristics of military software and of the DoD software environment impose several general design criteria on a suitable language:

- Reliability. The combination of extremely complex systems and life and death implications may be unique to the military. Language characteristics which promote the production of reliable software are weighted very highly.

- Modifiability. Most software costs (perhaps up to 90 percent) in embedded computer systems in the DoD are for software maintenance. Language features that contribute to the maintainability of reliable and efficient programs should have a major impact on software costs.
- Efficiency. Efficiency of object programs is a legitimate and sometimes critical concern in military applications. Physical limitations of military systems (e.g., aircraft) impose time and space limitations on computations. Software that cannot meet these constraints is worthless.

The desire for a common language that can be widely used throughout the DoD adds still more design criteria:

- Machine Independence. Over 200 computer models are known to be used currently in the DoD. The language must be sufficiently machine-independent that it can be made available on a variety of object machines.
- Practicality. The language must be sufficiently easy and inexpensive to implement that it will become widely available.
- Complete and Unambiguous Definition. The language must have a complete and unambiguous definition to ensure that software can be shared and to avoid incompatible implementations.
- Easily Accessible Support Software. The availability of useful and easily accessible support software is, of course, the ultimate technical goal of the common language effort, but the ease or difficulty in building such a support environment can be influenced strongly by the language characteristics.

The characteristics of a suitable language listed above were translated into a series of eight general requirements that constitute the first chapter of the *IRONMAN* requirements. These requirements are further expanded into specific constraints on the design of an appropriate language.

I. ASSUMPTIONS

The following assumptions were made in determining the technical requirements:

- Software reliability, object code efficiency, and modifiability of programs are more important than ease (i.e., terseness) of programming.
- The language must be as machine-independent as possible, but must be implementable on a wide variety of existing machines (i.e., as object machines).
- It is not necessary that all object machines be able to host the translators.
- Floating-point computations will be required only on object machines that have floating-point hardware.

Additional assumptions made with respect to specific requirements are discussed in the text.

II. TYPES OF ARITHMETIC

The common language is intended for a broad class of embedded computer applications that include sensor processing, real-time control, simulation, diagnostics, counting, record keeping, and display. All of these applications require numeric computation facilities in varying degrees of sophistication. The technical requirements for numeric computation facilities appear as Section 3 of the requirements document [1] and are reproduced in the Appendix.

Five kinds of arithmetic were considered: (a) integers, (b) fractions, (c) floating point, (d) scaled integers, and (e) scaled fractions. The four kinds other than floating point are particular forms of fixed-point arithmetic. The implications of each of these forms of arithmetic for the user and the translator are discussed in separate sections below.

A. INTEGERS

Counting is required in all digital applications, so some form of integer computation facility must be provided in every general-purpose program language. In addition, integers are often used to represent other data types that are not built into the language, for indexing arrays, and in all applications that have integer data.

Operations for integer arithmetic produce integer results from integer operands. Integer operations for addition, subtraction, multiplication, integer division, and remainder from integer division are usually needed. Integer operations can be computed and represented exactly in digital computers.

Because the representation of integers in computers must be finite, there is always some maximum range of integers that can be represented. The likelihood of errors from computations that exceed the available range is low because (a) the user usually knows the range of the (integer) values to be expected, and (b) the arithmetic hardware frequently provides a warning (i.e., fixed-point overflow) during execution whenever the result of a computation exceeds the maximum available range.

Most digital computers have hardware facilities for fixed-point arithmetic. That is, they provide arithmetic operations that assume that numbers are represented with a fixed number of digits and with the radix point either at the extreme right or extreme left of those digits (at the option of the user or translator writer). If one assumes that the radix is at the right, such machines implement integers directly.

A few computers have only floating-point hardware. In such machines, integers must be represented in floating-point format and the arithmetic must be executed using floating-point instructions. One special problem is associated with the implementation of integer arithmetic on floating-point hardware. If a result has more digits than can be represented exactly (i.e., integer range error), the usual floating-point convention is to discard the least-significant digit. This produces a result that only approximates the exact mathematical value and, therefore, can introduce unexpected errors in integer computations. A practical solution to the problem is for the hardware to provide a fixed-point overflow condition that is raised whenever a nonzero least-significant digit must be discarded to accommodate the value in the floating-point representation.

B. FRACTIONS

Fractions are numbers whose absolute values are less than one. Fractional computations are a form of arithmetic in which the operands and results of all arithmetic operations must be fractions. Because the representation of fractions in computers must be finite, there is always some maximum number of digits that can be represented. Less-significant digits are discarded. Thus, fractional arithmetic is most useful for computation involving data determined from physical measurements and other computations in which results need not be exact, but only computed to some finite precision (i.e., number of digits).

Fractional operations for addition, subtraction, multiplication, and division are usually needed. Each arithmetic operation produces a result that is rounded (or truncated) to the nearest representable value.

Fractional arithmetic is difficult to use because the user must scale all data so that it will be between -1 and 1. He must be sure when adding (or subtracting) that the absolute value of the sum (or difference) will be less than 1. He must be sure that the absolute value of the divisor is greater than the absolute value of the dividend on each division. He must determine the effective scale of final results in order to properly interpret output. All these conditions must be determined when the program is written (i.e., without knowledge of the exact data values).

Use of fractional arithmetic is further complicated because the effective precision of values is reduced whenever there are leading zeros in their representation. Ideally, the user must pick a scale for each value, X, that will maintain the magnitude of its representation in the range $B^{-1} \leq |X| < 1$ where B is the radix in which the number is represented.

Fractions are important because they can be used for inexact computations on noninteger values and because they can be implemented directly using the fixed-point hardware available on most digital computers (i.e., by interpreting the radix point to be on the left). Fractions correspond to the mantissa of numbers represented in scientific notation. The corresponding exponent (or characteristic) must be known, computed, and represented by the user as he writes a program.

C. FLOATING-POINT NUMBERS

Floating-point numbers are a finite approximation to numbers in scientific notation (i.e., computations in which exact results are not needed). Each floating-point number has a mantissa, M, an exponent, E, a radix, B, and a value $M \times B^E$. Because floating-point numbers have a finite representation, the mantissa has limited precision (i.e., number of digits in its representation) and the exponent is limited to some finite range. Floating-point values are often normalized (i.e., stored with a unique representation in which the absolute value of the mantissa, $|M|$, of all *nonzero* values is in the range $B^{-1} \leq |M| < 1$.

Floating-point arithmetic operations produce values that are close to the values that would be produced by applying the corresponding mathematical functions to the operands and then rounding the result to the nearest representable floating-point number.

Floating-point arithmetic is useful in any computation involving data representable in scientific notation. Floating point is easier and less error-prone in use than are fractions. Floating point is needed in computations where the ranges of values are not known at the time a program is written (i.e., where fractions cannot be used).

Floating point is not necessary if the exponent values are known during translation. Floating point requires more storage than fractions or integers for the same precision because the exponent as well as the mantissa must be stored during execution. Floating point requires special arithmetic hardware to manage the exponents during arithmetic operations and to maintain values in a normalized representation. This makes the hardware more expensive and frequently adds time to the execution of arithmetic operations (although these extra costs are often unimportant). Floating point is very expensive to use if implemented in software. The automatic rescaling by floating-point hardware not only makes floating point easier to use than other forms of arithmetic, but also makes floating point more dangerous to use. Because the scale management is automatic, the user is unaware of situations in which all significance (i.e., accuracy) is lost in a computation.

D. SCALED INTEGERS

Scaled-integer arithmetic, like integer arithmetic, is a system for exact numeric computation. A scaled integer is a product of an integer, M, and a scale, Δ. Any set of values can be represented exactly with the proper choice of Δ. Also, since any value can be approximated to any granularity by a suitable choice of Δ, scaled integers offer an alternative to fractions and floating-point numbers when noninteger computations are required.

The scale in the representation of a scaled integer must be specified as a constant when the program is written. This means that scaled integers cannot be used when the appropriate scale is unknown until execution (i.e., those cases in which floating point is required).

The translator can determine the scales of results during translation as a function of the scales of the operands. Consequently, only the (integer) mantissa, M, need be stored during execution and all operations during execution will be integer operations on the mantissas. Scaled-integer arithmetic can be exact because computing a mantissa involves only integers and computing the scale can be done symbolically because scales are processed only during translation and because (as will be seen later) only the prime factors and not the actual values of the scales are needed.

Applications that require exact computations on nonintegers can use scaled integers providing (a) the needed scales are known, and (b) the language allows the needed scales to be used. Applications that do not require exact computations can use scaled integers, providing the scales can be chosen to a sufficiently fine granularity to represent the needed precisions.

Scaled integers have been provided in many programming languages, usually with restrictions on the choice of scales. Many languages restrict the scales to a limited range of powers of 2 (i.e., $\Delta = 2^N$, with $-P \leq N \leq 0$ where P is the number of bits in the mantissa of the fixed-point representation of the object machine). Removing this restriction on N increases the time for neither compilation nor execution of a program, but it does greatly increase the ranges of values that can be represented (in fact, it permits ranges that cannot be represented in floating point).

Restricting Δ to powers of 2 does not affect the complexity of the scale computation. During execution, however, the cost of scale conversions can sometimes be reduced (assuming the object machine is binary) by using shifts instead of multiplying by powers of 2. If the language did not place restrictions on Δ, then a similar saving could be made whenever Δ is a power of the object machine radix. In many cases, however, multiplication would be required for scale conversion.

E. SCALED FRACTIONS

If integer scales can be managed automatically by compilers, then possibly so can scales of fractions. To automatically manage the scales of integers, it is necessary during translation to know only the maximum acceptable granularity and maximum ranges for the variables that are needed. To automatically determine the results scales for fractions during translation, it is necessary to know the expected values of variables, within a few (preferably one) powers of the radix (i.e., upper and lower bounds on the magnitude of the value of the variable). The lower bound is often difficult to predict. Finally, applications that require exact computation cannot use scaled fractions.

F. CHOICE OF TYPES

Some points from the above discussion are illustrated in Table 1, which shows the appropriateness of each of six types of arithmetic to each of four generic classes of applications. The columns correspond to integer computations, exact noninteger computations, inexact computations in which the upper and lower bounds on the magnitude of values are known when the program is written, and general inexact computations. A "YES" means that the arithmetic type can be used for the application. "NO" indicates that it would be very difficult or impossible to accomplish the computation with the designated numeric type. "DIFFICULT" indicates that although the computation can be accomplished, the use of this numeric type will be difficult, or inefficient.

TABLE 1. SUITABILITY OF NUMERIC TYPES TO APPLICATIONS

	Application			
	Exact Results Needed		Exact Results Not Needed	
Type	On Integers	On Nonintegers	With Predictable Values	Without Predictable Values
Integer	Yes	No	No	No
Fraction	No	No	Difficult	No
Floating Point	Some Problems	No	Yes	Yes
Scaled Integer $\Delta = 2^N$	Yes	No, except for powers of 2	Yes	No
Scaled Integer Unrestricted Δ	Yes	Yes	Yes	No
Scaled Fractions	No	No	Yes	No

The numeric types chosen for the common language are integer and fixed point in the form of scaled integers, and floating point. The main considerations for each type are given below:

- Integer. Integers are not needed if scaled integers are provided. On the other hand, integers are so frequently needed and are sufficiently simpler in their use than are scaled integers, that integers should be either a distinct type or a special case of scaled integers.

- Fractions. Fractions provide no power that is not provided with more simplicity by the other types. Fractions should not be included as a type in the common language.

- Floating Point. Floating point is essential for some applications and, therefore, should be provided by the language. Because floating-point hardware is not available on all object machines, floating point is not always a viable alternative to scaled integers or fractions when the magnitude of values are known at translation time. Floating point is required, but its inclusion does not alleviate the need for some form of automatically scaled arithmetic. Also, because floating point is so expensive if implemented in software, it need not be provided in object machines that do not have floating-point hardware (i.e., we assume that if the application requires floating point, it will use a machine with floating-point hardware).

- Scaled Integers. Scaled integers are needed for exact computations on nonintegers and applications should not be restricted to powers of 2 (in particular, powers of 10 are needed). Scaled integers also offer an acceptable alternative to floating point when the magnitude and scale of values are predictable at the time of translation. Scaled integers also permit wider precision of values than is normally provided by a floating-point facility. A fixed-point type in the form of scaled integers should be provided by the language.

- Scaled Fractions. Scaled fractions provide few advantages over scaled integers and cannot be used when exact results are required. Thus, scaled fractions should not be built into the language.

III. FIXED-POINT AND INTEGER TYPE

The language shall provide a fixed-point and integer type (3-1Aa*). Integers are intended for all integer computations including counting, denoting the ordinality of a particular object in a set of similar objects (i.e., indexing), and representing elements (i.e., atoms) of data types that are not built into the language. Fixed-point numbers are intended for all numeric applications that involve values other than integers and require exact results, for numeric computations in which the ranges of values do not vary dynamically, and as a substitute for floating point when floating-point hardware is not available.

A fixed-point and integer type can be implemented efficiently on most existing digital computers. Each fixed-point or integer variable in a program is required to have a range, R_1 to R_2, and a positive scale, Δ, chosen and specified in the program by the user. All values, X, that must be represented in such a variable must be within the specified range (i.e., $R_1 \leq X \leq R_2$) and must be integral multiples of the scale. For integers, the scale is 1. Because the scale of each variable is constant during execution (3-1G), a value of the variable can be represented, in the scale, Δ, as a single integer M where $M \times \Delta$ is the value. The amount of storage required for the variable (e.g., word width) is just that needed to store any integer M where $R_1/\Delta \leq M \leq R_2/\Delta$.

*Hereafter references to the *Revised IRONMAN* are indicated by the requirement number in parentheses. In some cases the requirement number is followed by a lower-case letter to indicate the sentence.

A. RANGE

The range of each fixed-point and integer variable must be specified in programs (3-1Ca) and indicates that all legal values lie within the limits of the range. Range information can be helpful in understanding and maintaining programs, it is needed by the translater to determine how much space must be allocated to a variable. It is also a form of assertion which can be used to aid proofs of correctness or which can be checked automatically during execution. If the value lies outside the range, a range error is said to have occurred.

A range specification shall be interpreted by translators as the minimum range to be implemented (3-1Cb). That is, every implementation of variables with a particular specified range will support that range on the designated object machine (or state that the variable cannot be represented). However, efficiency considerations may dictate that a desirable implementation should support a wider range than that specified. This wider range is known as the implemented range and may vary from object machine to object machine, or within the same object machine according to optimization needs. By permitting a variable to have unused states in its implementation, a language allows the translator to select the most efficient implementation that will not adversely affect the program correctness. For simple variables, a full word is usually the most efficient representation, regardless of the specified range, so that the range specification simply gives the translator a way to determine whether the object machine word length is adequate. Arrays, however, can often be represented more compactly, and without loss of execution time, where it is known that a partial word representation will be adequate.

The range of each fixed-point and integer variable must be determinable at the time of its allocation (3-1Ca). For most variables, the specified range will be constant and, therefore, can be determined during translation. In most object machines, the only efficient representation will be as a full word, so the actual range need not be known during translation. Indeed, there are cases where it is desirable to delay binding of the specified range until scope entry (e.g., within a procedure, a local variable that is used to index an array parameter should have its range bound to the array subscript range which may vary from call to call). In such cases, it is safe for the translator to use the largest representation that is used elsewhere in the program (and which typically will be the largest efficient integer representation). An optimizing compiler might implement the minimum range that will satisfy all the actual ranges that can occur during execution.

The maximum range of intermediate results in an expression can be determined automatically from the operations and the range of the operands. Each operation, however, tends to expand the range of the result so that in an assignment statement or similar context, the maximum computed range will almost always exceed the range of the variable being assigned (e.g., $X \leftarrow X+1$ contains a potential range error because the computed range of X+1 will not be within the range of X).

One way to eliminate range errors is to require the use of explicit range conversion operations (such as modulo) that will guarantee that values will be in range. Because such range conversion operations would have to be numerous in programs, they would detract from readability and would reduce efficiency. Hence, explicit conversion operations shall not be required between numeric ranges (3-1Cc).

Another way to eliminate range errors is to use implicit range-conversion operations. There are many choices: reduce each value modulo the specified range, reduce each value modulo the implemented range, replace any value that is out of range by the nearest extreme value of the range, replace any value that is out of range by a designated constant, etc. Any of these choices can cause unexpected results, most would add to execution costs, and none would be appropriate for all applications. Implicit range conversions modulo the implemented range would be most efficient and has been a traditional method because they can be implemented directly by most integer and floating-point hardware. Because word lengths are not standardized, however, such a choice would require that the semantics of correct programs be machine-dependent.

A third alternative is to raise an exception during execution whenever a range error occurs. This is the approach taken by *IRONMAN*. The language will support a mechanism, exception handling, whereby the user can specify, in the program, the appropriate responses to different errors detected during execution of the program. There shall be exceptions during execution whenever a value exceeds either the specified range of a variable or the implemented range (10Ba). However, tests to determine the presence of values outside the specified range can be expensive during execution because they must be done in software on current machines. In some cases, the translator will be able to prove that a range error cannot occur and, thus, can safely eliminate the corresponding test. Tests to detect values outside the implemented range are often inexpensive because they can be implemented directly using the integer or fixed-point overflow interrupt of the object-machine hardware. A programmer option that can improve efficiency without much loss of safety is to suppress detection of errors on the specified range (10Ga) but not on the implemented range. The effect on the implementation should be the same as if the specified range were extended to exactly the implemented range. This does not affect the semantics of correctly written programs, regardless of object machine, but it can lead to different results on different hardware (e.g., different word widths, different overflow detection mechanisms) for incorrect programs.

A closely related problem arises in determining the range of intermediate results. Although the variables of a program may have ranges that are efficiently implementable in the object machine, the worst-case ranges (as computed from the ranges of the operands) for intermediate results in an expression may exceed any range that can be implemented efficiently without loss of information. In correctly written programs, the actual intermediate results will tend to be within efficiently implementable ranges.

The language designer has several choices when there is a potential error on the implemented range of an intermediate result. The language might require the translator to (a) change the scale of the intermediate result so that information is lost from the least-significant digits, (b) give an error during translation, (c) provide a safe but inefficient implementation, or (d) provide an efficient implementation with an exception during execution if the actual value does exceed the implemented range.

The first approach should be discounted because it would lose information in a fixed-point facility that is to provide exact results. The second case will give a translation error in the frequent case in which the actual values can be efficiently represented but the potential values exceed any efficiently implemented range. The third approach is theoretically best because it is safe and does not complicate the language. It may, however, complicate the translator and will unnecessarily increase the execution costs of programs in which the potential range exceeds the maximum efficiently implementable range, but the actual values do not.

The last (fourth) approach is also safe because range errors will be detected during execution. Its primary advantage is that no unnecessary execution costs are incurred for the usual situation in which the actual data does not cause a range error. Its main disadvantage is that it places an extra burden on the user when the actual values exceed the implemented range.

The translator should warn the user (during translation) of those situations where there is a potential range error (i.e., a warning that the implemented range does not cover the full (implicitly specified) range of an intermediate result). A variation of the last approach is for the translator to implement an automatic (correct but inefficient) reevaluation of the expression in response to the range exception. The language definition should specify which semantics will be given. If the fourth-case semantics is the choice, translators can use the third or either of the fourth case implementations.

B. SCALE

Every value of a fixed-point variable is an integral multiple of the variable's scale or step size. The step size of each variable must be chosen individually to match the needs of the application, but can be fixed during translation (3-1G) to permit the most efficient implementation. Because the scale of each variable is known during translation and is unalterable during execution, only the integer need be stored during translation.

Scale conversions take place when a value has a scale which differs from that of the variable to which it is being assigned. Conversions can be divided into two groups, those for which the values can be represented exactly in the target scale (i.e., of the variable), and those which cannot be so represented. As an example of the first, one might want to assign the value of a variable that has a scale of 1/2 (i.e., values that are a multiple of 1/2) to a variable of scale 1/4. In such a case (i.e., any case for which the scale of the value is an integral multiple of the scale of the variable), the value can be represented exactly in the variable and such assignment should be allowed without an explicit scale conversion operation in the source program, since the actual value is unchanged. With the most obvious representation for these scales (i.e., radix points one and two binary digits from the least significant end of the word for scales 1/2 and 1/4, respectively) a one bit shift left would be required as part of the assignment. In general (i.e., for scales that differ by an integer that is not a power of the radix of the object machine), implicit scale conversions will require an integer multiply instead of a left shift.

Many implicit scale conversions can be eliminated during translation by using less compact representations. In the above example, the value of scale 1/2 could be kept (or computed) in the scale 1/4, and a zero kept in the (additional) least-significant bit of the mantissa. This method can eliminate execution time for implicit scale conversions, but reduces the range of the values that can be represented and may add to the cost of multiplication to avoid unnecessary (implemented) range errors in intermediate results.

Fixed-point and integer values are intended for computations that require exact results or require a detail of control that can be obtained only with an exact computation (3-1Fa). On the other hand, it is sometimes necessary to convert numeric values to a scale which cannot represent them exactly. That is, some other (arithmetically close) value in the desired scale must be used instead of the computed value. To avoid implicit changes in value, the language shall require explicit scale conversion operations whenever the abstract value may be changed (3-1Hb).

There shall be built-in operations for conversion between fixed-point (and integer) scale factors (3-1Ha). Rescaling can be done by either truncation or rounding, but which is appropriate for a given application cannot be determined automatically by the translator. Consequently, there shall be no implicit truncation or rounding in fixed-point and integer computations (3-1Fb).

There are two common definitions of truncation, towards minus infinity (i.e., the greatest value in the target scale not greater than the source value), and towards zero (i.e., in the target scale, the value that is farthest from zero but is not farther away than is the source value). The major invariant property of the first is:

$$\mathrm{TRUNCATE}(X+n\Delta,\Delta) = \mathrm{TRUNCATE}(X,\Delta)+n,$$

while the second has

$$\mathrm{TRUNCATE}(-X,\Delta) = -\mathrm{TRUNCATE}(X,\Delta).$$

It is possible, but nontrivial, to convert one definition to the other. It may be desirable that any choice between the two be compatible with the definition of integer divide and the remainder operation (see III-C, below) which corresponds to the decision between whether the sign of the remainder is the same as the sign of the divisor or the dividend. (An always positive remainder has no equivalent, reasonable definition for truncation.)

Rounding is the process of obtaining the "nearest" value in the target scale. With the first definition of truncation, rounding a value X to scale Δ can be expressed as

$$\mathrm{ROUND}(X,\Delta) = \mathrm{TRUNCATE}(X+\Delta/2,\Delta),$$

whereas the second definition leads to

$$\mathrm{ROUND}(X,\Delta) = \mathrm{TRUNCATE}(X+\mathrm{sign}(X)\times\Delta/2,\Delta).$$

These two definitions have their hardware analogue in two's complement and one's complement implementations of arithmetic. However, the language definition must choose one which translators must then implement, regardless of the object machine. A firm resolution of this issue in languages might lead to greater uniformity in hardware design.

The choice between the two is not clear-cut, but several factors can be noted. In particular, the invariant of the first has greater applicability than that of the second because it is true for any integer n rather than just -1. Where existing high-level languages have taken a position, they have chosen the first definition (e.g., entier in the Algol's and floor in APL). On machines that directly implement truncation toward zero, truncation toward minus infinity is relatively more expensive to implement. In most cases, however, the actual value will be positive (in which case the two forms of truncation are indistinguishable) and will be known to be positive by the translator (because of range specifications), so that whichever hardware implementation is available can be used. Thus, it appears that there are valid arguments both ways. To avoid ambiguity , however, the language must define truncation and division on negative arguments.

Truncating and rounding are always to some scale, Δ. To avoid the introduction of scales that vary during execution, the desired scale in a scale-conversion operation must evaluate to a constant during translation.

In some contexts, the actual parameter specifying the scale in an explicit scale conversion will be redundant and may be omitted. Such contexts include the right hand sides of assignment statements (where the scale is that of the variable, for example, X←ROUND(Y)), actual parameters (where the scale is that of the formal parameter), and array indices (where the scale is one).

Although explicit rescaling is always safe, it is unnecessary whenever it is known by other means that an actual value (but not all values of its scale) is exactly representable in the desired scale. For example, a value of scale 1/4 is computed to be assigned to a variable of scale 1/2, but it is known that the actual value in this case will be a multiple of 1/2. In such a case, neither truncation nor rounding will alter the value but might unnecessarily increase the execution costs. Unnecessary scale-conversion operations can be omitted from the object program wherever the translator can determine that the optimization is safe.

What fixed-point scales should be allowed by a language? Because fixed-point computations are exact, the scale must be exact; fixed-point scales cannot be floating-point values. Several possible choices for the scales that a language should allow have been suggested: negative powers of 2, any power of 2, powers of 10, integers and their reciprocals, any rational, and any real number. Powers of 2 and 10 (both positive and negative) and their products are most useful. In special situations, other rational scales might be needed. Irrational numbers are sometimes useful (e.g., a variable might contain only multiples of pi). Because scale computations are done entirely at translation time, they cannot add to the execution costs. The choice of the scale affects execution only during scale conversions; rescaling by powers of the object machine radix can be done by shifting, while the rest require multiplication.

Once powers of 2 and 10 are allowed, allowing any integer or reciprocal of an integer as a scale will not add to the complexity of the language or its translators. Similarly, once powers of 2 and 10 and their products are allowed, allowing any rational scale will not increase the complexity of the language or its translators (the computational details are discussed in Sections C and D below). Irrational scales, on the other hand, are seldom useful, would add to the complexity of translators (i.e., such scales would have to be treated symbolically), but are not expensive in execution (i.e., the methods of Sections C and D also apply to irrational scales).

C. FIXED-POINT ARITHMETIC

If all rational scales are allowed for fixed-point values, then any rational number can be represented exactly. There will be one abstract value corresponding to each fixed-point representation (i.e., mantissa and scale). Information need not be lost in fixed-point computations. Fixed-point arithmetic should be exact with each operation producing a fixed-point representation for the abstract value that is the exact result of applying the corresponding mathematical operation to the abstract values represented by the operands (3-1Fa).

Several fixed-point and integer arithmetic operations are needed: addition, subtraction, multiplication, integer division, remainder, and division (3-1Bb, 3-1Ha). All of these operations have well-defined meanings, are usually included in programming languages, are useful in any language that requires exact arithmetic, and can be efficiently implemented.

The individual operations are discussed below. In each case, operands will be designated as $X_i\Delta_i$ i=1,2 where X_i is the integer mantissa of the internal representation and Δ_i is the rational number that is the specified scale and is processed entirely during translation. The symbols and abbreviations used below are for exposition in this paper and do not represent a preference for a common language.

Addition and subtraction are least expensive when the scale of their operands are identical: $X_1\Delta_1 \pm X_2\Delta_1 = (X_1 \pm X_2)\Delta_1$. When the scale of the operands differ, they can be (implicitly) converted to a common scale without altering their abstract values (3-1Hb).

Let this common scale be Δ_3. Then, $X_1\Delta_1 = X_1(\Delta_1/\Delta_3)\Delta_3$ where Δ_3 has been chosen so that Δ_1/Δ_3 is an integer. Thus, addition and subtraction might be accomplished as

$$\begin{aligned} X_1\Delta_1 \pm X_2\Delta_2 &= X_1(\Delta_1/\Delta_3)\Delta_3 \pm X_2(\Delta_2/\Delta_3)\Delta_3 \\ &= (X_1\Delta_1/\Delta_3 \pm X_2\Delta_2/\Delta_3)\Delta_3. \end{aligned}$$

For the most efficient representation of the sum or difference, Δ_3 should be as large as possible. Δ_1/Δ_3 and Δ_2/Δ_3 are thus the smallest pair of integer multipliers, for exact addition and subtraction, between values in the scales Δ_1 and Δ_2. They have no common integral factor (i.e., greatest common divisor, GCD) greater than unity (otherwise it would have been absorbed into Δ_3).

It is neither trivial nor difficult for the translator to evaluate Δ_3. Δ_i can be represented as the quotient of the two coprime (i.e., relatively prime) integers, P_i and Q_i (i.e., $\Delta_i=P_i/Q_i$, and $\mathrm{GCD}(P_i,Q_i) = 1$). Thus,

$$X_1\Delta_1 \pm X_2\Delta_2 = X_1\frac{P_1}{Q_1} \pm X_2\frac{P_2}{Q_2} = \frac{X_1P_1Q_2 \pm X_2P_2Q_1}{Q_1Q_2} .$$

$\Delta_3=P_3/Q_3$ is the rational factor common to both halves of this expression and is $\frac{\mathrm{GCD}(P_1Q_2,P_2Q_1)}{Q_1Q_2}$. Since P_i and Q_i are coprime, this reduces to $\frac{\mathrm{GCD}(P_1,P_2) \times \mathrm{GCD}(Q_1,Q_2)}{Q_1Q_2}$ which further reduces to $\frac{\mathrm{GCD}(P_1,P_2)}{\mathrm{LCM}(Q_1,Q_2)}$ where LCM is the least common multiple. Thus, P_3 and Q_3 are the smallest integers such that $\frac{P_3}{Q_3} = \frac{\mathrm{GCD}(P_1,P_2)}{\mathrm{LCM}(Q_1,Q_2)}$ and the required addition or subtraction is achieved by multiplying X_1 by $\frac{P_1}{Q_1} \times \frac{\mathrm{LCM}(Q_1,Q_2)}{\mathrm{GCD}(P_1,P_2)}$ and X_2 by $\frac{P_2}{Q_2} \times \frac{\mathrm{LCM}(Q_1,Q_2)}{\mathrm{GCD}(P_1,P_2)}$. It will be observed that both reduce to integers (i.e., only multiplicative rescaling is required) since $\mathrm{GCD}(P_1,P_2)$ is an integral factor of both P_1 and P_2, and also that Q_1 and Q_2 both divide $\mathrm{LCM}(Q_1,Q_2)$ exactly. $\mathrm{GCD}(P_1,P_2)$ and $\mathrm{LCM}(Q_1,Q_2)$ are coprime since P_1 and Q_1, and also P_2 and Q_2, are coprime, and hence any factor of $\mathrm{GCD}(P_1,P_2)$ is a factor of P_1 and P_2 and cannot, therefore, be a factor of Q_1 or Q_2, and hence not of $\mathrm{LCM}(Q_1,Q_2)$. Thus, $P_3 = \mathrm{GCD}(P_1,P_2)$ and $Q_3 = \mathrm{LCM}(Q_1,Q_2)$. By analogy with integers, it will be useful to treat Δ_3 as the $\mathrm{GCD}(\Delta_1,\Delta_2)$ in the rest of this paper.

It will be noted that, if $P_1 = P_2 = 1$, then $\Delta_3 = \frac{1}{LCM(Q_1,Q_2)}$ (e.g., 1/2's and 1/3's must be added in 1/6's, which is intuitively correct). Also, if the operands have the same scale (i.e., $P_1 = P_2$ and $Q_1 = Q_2$), then $\Delta_3 = P_1/Q_1$ so that addition and subtraction is performed directly in the scale of the arguments.

At most, one multiplication is required during execution to convert each operand. If an operand's scale is a multiple of the scale of the other operand (i.e., $\Delta_1 = GCD(\Delta_1,\Delta_2)$ or $\Delta_2 = GCD(\Delta_1,\Delta_2)$), only one conversion is necessary. In the usual case for addition and subtraction, the operands' scales will be identical and no scale conversion will be needed. When several incompatible scales are added or subtracted in a single expression, each operand can be converted to the final result's scale and no additional scale conversions will be needed for intermediate results.

It should be noted that implementation of scale management by the translator is eased if it is remembered that $GCD(P,Q) \times LCM(P,Q) = P \times Q$. There are reasonably efficient "arithmetic" implementations for the evaluation of GCD(P,Q) based on GCD(P,Q) = GCD(P-Q,Q). Alternatively, a "list processing" implementation of scale factors in the translator could be considered (e.g., $12 = 2^2 \times 3^1$ could be represented as <<2, 2>, <3, 1>>), in which case GCD, LCM, and multiplication become list-processing merges (i.e., min, max, and pointwise addition, respectively). This approach enables the numerator and denominator of the scales to be held together (e.g., 50/27 = <<2, 1>, <3, -3>, <5, 2>>).

Multiplication never requires rescaling of the operands. The scale of the product is the product of the operand's scales: $X_1\Delta_1 \times X_2\Delta_2 = (X_1 \times X_2)(\Delta_1\Delta_2)$.

Integer division is an exact division operation that can be defined over all fixed-point values and in all contexts. The result of the integer division, $X \div Y$, can be defined as the largest integer that is not greater than the true quotient of X divided by Y. Integer division by zero is undefined and should cause an exception. As thus defined, integer division has the property: $(X+nY) \div Y = X \div Y+n$ for any integer n and corresponds to truncation towards minus infinity.

A remainder (or modulo or residue) operation, MOD, can be associated with integer division and is normally defined as the difference between the dividend and the product of the divisor and the integer quotient: $\mathrm{MOD}(X,Y) = X-(X \div Y) \times Y$. With the above definitions, $X \div Y \leq X/Y$ and the sign of the remainder is the sign of the divisor.

An alternative definition of integer divide is truncation towards zero of the true quotient, and has the properties that

$$\mathrm{ABS}(X \div Y) = \mathrm{ABS}(X) \div \mathrm{ABS}(Y),$$

and

$$\mathrm{ABS}(X \ \mathrm{MOD}\ Y) = \mathrm{ABS}(X) \ \mathrm{MOD}\ \mathrm{ABS}(Y).$$

In this case, the sign of the remainder is the sign of the dividend.

The scale of the remainder will be the same as the scale of the difference between the operands. Thus, the scale of $(X_1\Delta_1 \div X_2\Delta_2) \times X_2\Delta_2+\mathrm{MOD}(X_1\Delta_1,X_2\Delta_2)$ will be $\mathrm{GCD}(1 \times \Delta_2,\mathrm{GCD}(\Delta_1,\Delta_2))$ which is $\mathrm{GCD}(\Delta_1,\Delta_2)$, even though it is exactly representable in scale Δ_1. This implies that $X = (X \div Y) \times Y+\mathrm{MOD}(X,Y)$ will be true, but that $X \leftarrow (X \div Y) \times Y+\mathrm{MOD}(X,Y)$ will be interpreted as a scaling error when $\mathrm{GCD}(\Delta_1,\Delta_2) \neq \Delta_1$.

Full division of fixed-point numbers is also useful and will always produce a result that can be represented as a fixed-point number. However, in general, the result scale for division cannot be determined until execution because it depends on the mantissa of the divisor. Fixed-point division can be implemented in a language with static scales if the user is required to specify the desired scale for the result. That is, the language might permit expressions of the form: TRUNCATE(X/Y,S) or ROUND(X/Y,S) where the scale S is a constant. TRUNCATE($X_1\Delta_1/X_2\Delta_2,\Delta_3$) can then be implemented as $(X_1 m \div X_2 n)\Delta_3$ where m and n are coprime integers such that $m/n = \Delta_1/(\Delta_2\Delta_3)$. Similarly, ROUND($X_1\Delta_1/X_2\Delta_2,\Delta_3$) can be implemented as $(((2mX_1 \div nX_2)+1)\div 2)\Delta_3$. A user view of such a division might be floating-point division with conversion of the result to fixed point (3-1Bb).

As with scale-conversion operations (Section III-B above), in certain contexts the explicit scale for fixed-point division will be redundant and may be omitted (e.g., when the quotient is to be assigned to a variable of known scale).

Another special case of fixed-point conversion is division by a constant K. The division $(X_1\Delta_1)/K$ yields $X_1(\Delta_1/K)$ where the division can be performed entirely during translation and results in a reinterpretation of the associated scale during execution.

Other operations, such as negation and absolute value, are also useful. On machines with only one representation for zero (e.g., 2's complement, 10's complement) the number of negative fixed-point values that can be represented in any given scale is one greater than the number of positive values, so both negation and absolute value can cause an exception by producing a positive value that exceeds the implemented range.

D. OTHER FIXED-POINT OPERATIONS

Relational operations are needed in all general-purpose programming languages. For completeness and conformity with existing languages, the six relational operations $< \leq = \neq \geq >$, are needed (3-1Bc). The relational operations on fixed-point values can have a meaning consistent with the normal mathematical definition between their corresponding abstract values. In principle, the scale of the comparison should be that of the difference between the operands. This can cause a considerable widening of the representation if the scales are disparate, leading to the risk of overflow exceptions or the generation of inefficient and/or unnecessarily complex code sequences. In practice, the translator can determine from consideration of the ranges whether this exception may occur and issue a suitable warning. It should not be considered contrary to the language definition for the translator to implement comparisons by methods which give correct answers efficiently in most cases, provided no program is allowed to continue if the correct result cannot be obtained.

Floating-point values are sometimes used as arguments in fixed-point computations. Because such uses constitute a tightening in the interpretation of the values (from inexact to exact), there should be built-in operations for conversion from floating point to fixed point (3-1Ba). In order to convert from floating point to fixed point, the desired scale must be known. Because the scale cannot, in general, be determined from the context, the conversion operation should have a parameter specifying the desired scale. There are again, however, certain contexts, such as assignment of the value to a variable, where the scale is implicitly specified, so the explicit parameter may be omitted. Because floating-point values cannot usually be represented exactly in a given fixed-point scale, conversion from floating point to fixed point may change the abstract value that is represented. Thus, there should be no implicit conversions from floating point to fixed point (3Ba). For consistency with scale-conversion operations, both truncation and rounding conversion operators should be available. As with scale conversions, the scale argument must be a constant at the time of translation.

To facilitate the writing of generic definitions, the language should provide a translation-time function that can be applied to any fixed-point variable or value to obtain the maximum (i.e., specified) scale with which the variable or expression can be represented (in some cases, the translator might use a smaller scale in the actual representation to reduce execution time) (12Da). A scale operator would be most useful in generic definitions where it could be used to declare a variable to have the same scale as that of an actual parameter. A scale operator could also be used to improve the readability of explicit scale conversions. For example, rounding X to the scale of Y might be written as: ROUND(X,SCALE(Y)). Because the scales of fixed-point variables are bound during translation, the scale operator can always be evaluated during translation. Inclusion of a scale operator will allow the definition of standard mathematical functions on fixed-point values without prior knowledge of the particular scales that are appropriate to a given application.

E. FIXED-POINT LITERALS

There shall be built-in numeric literals (2Ga). Numeric literals are needed to designate numeric constants in programs. Any fixed-point constant, including literals, can be represented exactly in a variety of scales. The larger the scale used, the smaller the space required for intermediate results in expressions that contain the constant. The maximum scale for a constant is, of course, the constant itself (with a mantissa value of 1). Thus, all literals can be implemented exactly and neither the language nor programs should restrict the scales used to represent literals; the translator can do this by treating literals as if their values were their scale. Implicit scale conversions, that are needed to perform subsequent operations (such as addition) efficiently, can be done during translation.

F. NUMERIC INPUT-OUTPUT

This section is concerned with the physical representations that will be associated with numeric data on external storage files, with how translators will learn the external representations, and with the operations that convert to and from symbolic representations for display and input, respectively. The *IRONMAN* does not directly address the problem of input and output of numeric data.

Because the formats of records, the precision of floating-point numbers, and the scales of fixed-point numbers are bound at translation (3-3Ba, 3-3Gb, 3-1Da, 3-1G), programs will be able to read and write records using logical structures known to the translator. In theory, the physical format of a record could be carried with a file and processed during execution. In practice, there is no reason to dynamically vary the physical representation (for a given logical representation), and, therefore, the physical representation can be given to the translator rather than to the object program.

In some cases, files must be read from or written to foreign systems whose format conventions would be incompatible with any fixed convention established by a language design or translator. Consequently, the language must provide a facility for user specification of the physical representation of records (11Aa). Such a specification, although distinct from the corresponding logical specification, must be compatible with the logical specification. The facility for physical specifications must be sufficiently detailed, unambiguous, machine independent, and translator independent that the same description can be given to translators for different object machines so that the resulting programs will be able to write files of the designated record format which can be read correctly on another system.

A common language that is intended for a wide variety of object machines with different word lengths and numeric representations cannot dictate any convention about the representation of numeric values. Properties, such as the implemented scale, the implemented precision, sign magnitude versus two's complement versus one's complement, radix, exponent range and representation, and the presence of "don't care" fields within numeric representations, must be specifiable. In practice, most systems would use only those that are compatible with their object-machine characteristics.

Input-output operations can be kept simple and efficient by restricting them to data structures already present in programs. That is, it should be possible to read and write only those records whose internal and external representations are identical. The input and output operations need not do format or representation conversions. Such conversions, if needed, must be accomplished by operations between records or fields of records within the program. If the corresponding fields are incompatible (e.g., in radix), appropriate explicit conversion operations will be required. Such conversion need not be built into the language but must be definable in the language.

Although there must be a mechanism for specifying the physical representation of a record, its use will often unnecessarily burden the user, may preclude more efficient representations that could be automatically determined by the translator, and adds to the complexity (and, therefore, error proneness) of programs. Thus, specifications for the representations of records should be optional (11Ad).

For files that are written and read entirely within one program, the physical representation of records (including numeric fields) can be chosen and managed by the translator. With additional bookkeeping, representations could be managed safely for related programs that are developed together in a common host system. In the latter case, the translator that defines the representation must encode the representation in some formal notation that can be maintained by the host system and accessed later as other programs are translated. If the formal notation is identical to that used for specifying representations in programs, it could also be taken from the system in symbolic form and inserted into programs that are compiled on other host systems.

Input and output of numeric data in symbolic form require two lower-level facilities: a mechanism to input and output characters, and a mechanism to convert between symbolic and internal numeric representations of data. The standard input-output operations will provide for reading and writing of character strings (8A, 8B). Conversion between numeric and symbolic representations of numbers should be definable within the language and made available as standard library routines. If the symbolic forms are restricted to decimal numbers, it may be necessary to restrict the conversion operations to numeric values that have scales that are powers of 10.

The only restriction imposed by the *IRONMAN* is that identical symbolic representations of numbers will produce the same internal values, whether they are used as literals in programs or are read as data using the standard conversion routines during execution (2Gb). This means that all translators must use the standard conversion routines or ones that are functionally equivalent.

IV. FLOATING-POINT TYPE

The language shall provide a floating-point type (3-1Aa). Floating point is intended for those applications in which the results need be precise only to some specified number of significant digits and for which there is wide dynamic variation in the range of values. Most engineering and scientific calculations can be done conveniently using floating point, as can calculations that require inputs from sensors or outputs to control devices. Many of these calculations can, however, be done equally well using fixed point.

If floating point is used in a program for an object machine that does not have floating-point hardware, the translator should give a warning that the floating-point computation (which must, therefore, be implemented in software) will be unusually expensive in execution (13Dc). It is anticipated that the standard library will include a software definition of floating-point data and operations (without the usual hardware restrictions on maximum precision).

A floating-point variable has a fractional mantissa, M, (i.e., $-1 < M < 1$), a positive integer base, B, an exponent, E, a range, R_1 to R_2, and a precision, P. The value of such a number is $M \times B^E$. Only the mantissa and exponent need to occupy storage in the object machine. The base is constant and is determined by the floating-point hardware implementation. The precision is the number of significant digits in the mantissa. Values that have more than P nonzero leading digits must be approximated. In many floating-point notations, each nonzero floating-point value is normalized to eliminate leading zeroes from the mantissa, that is, the mantissa is shifted left until $-B^{-1} \geq M$ or $M \geq B^{-1}$ and E is decremented by the number of digits shifted. The range specification designates upper and lower bounds on the values that are to occupy a variable (3-1Cb), that is, $R_1 \leq M \times B^E \leq R_2$. Because $-1 < M < 1$, it follows that $E < \log_B(\max(\mathrm{abs}(R_1),\mathrm{abs}(R_2)))$ and the range of a floating-point variable is an indirect specification of the range of the exponent.

This chapter is concerned with floating-point computation as viewed by the user, the language designer, the translator writer, and the hardware designer. The user of a floating-point system wants (a) a floating-point representation that has sufficient range and precision to satisfy the needs of his application, (b) computational rules that minimize the loss of significance (i.e., accuracy) in his computations, and (c) floating-point operations for which the significance of the results is predictable from the significance of the arguments.

It is the responsibility of the language designer to provide language facilities and conventions that will supply sufficient information that the translator will know what ranges of values are expected in a program and what precisions are needed. It is the responsibility of the translator writer to ensure that (a) the implementation meets the range and precision requirements of the program, (b) that the implementation does not unnecessarily cause loss of significance, and (c) that the implementation is as efficient as is possible using the available floating-point hardware. It is the responsibility of the hardware designer (a) to ensure that the floating-point hardware does not cause unnecessary loss of significance and (b) to ensure that error propagation from round-off is predictable.

A. RANGE

The range of a floating-point variable must be specified in the program and be determinable at the time of its allocation (3-1Ca). The user will view such specifications as the maximum ranges needed for the computation. The translator writer will view such specifications as the minimum ranges to be implemented (3-1Cb). Explicit conversion operations shall not be required between numeric ranges (3-1Cc). There shall be an exception during execution whenever a value exceeds either the specified range of a variable or the implemented range (10Ba). It shall be possible to suppress individually the detection of an exception within a given scope (10Ga). Both for consistency and because the same considerations (III-A) apply, these requirements are identical to those for the fixed-point and integer type. Special considerations for floating point are discussed below.

The range of a floating-point variable affects the space required for the exponent but not the mantissa. Because most floating-point hardware provides only a single choice for the exponent range, range specifications can be used to determine whether the available exponent range is adequate, but usually do not affect the implementation. Consequently, delayed binding of the range will not adversely affect the implementation.

Detection of values outside the implemented range is inexpensive because such errors correspond exactly to the floating-point overflow interrupt in most hardware implementations.

B. PRECISION

The significance or accuracy of data and computational results is a primary concern to anyone using a floating-point facility. A floating-point implementation, however, deals only with the precision of data and results. The precision of a variable is the number of digits to be allocated to its mantissa. The precision of an expression is the number of digits to be computed when evaluating the expression. If, for example, a variable, X, has precision, P, P+1 digits may be required to compute the sum X+X without loss of information, and 2×P digits may be needed for the product X×X. The precision that is needed to avoid loss of significance depends on the significance of individual arguments, the operations involved, and the desired significance of the results; the precision that is needed may vary from variable to variable and from expression to expression. Thus, the user should be able to specify a precision for each floating-point variable and expression in a program (3-1Da). This specification shall be interpreted by the user as an upper bound on the significance provided by his data or expected in his computation results. The specification shall be interpreted by the translator as a lower bound on the precision to be provided in the implementation.

The language shall require explicit specification of the precision for floating-point variables (3-1Db). Because the appropriate precision for a variable depends on the characteristics of the applications, no one choice for a default (i.e., implicitly specified) precision would satisfy most cases (i.e., reflect common usage as in 1Ce). For an expression or subexpression, there can, however, be a default precision that is determined from the precisions specified for its arguments. In an assignment statement, the significance of the value after assignment cannot be greater than the precision of the variable; thus, the precision of the right-hand-side expression need be no greater than that of the variable being assigned. A larger precision would require additional computation for digits that would then be thrown away.

Ideally, the default precision for a subexpression would be the minimum precision that would produce the maximum obtainable significance (up to the specified precision) for the result of the expression without unnecessary computation. Table 2 shows the minimum significance that the arguments must have to guarantee that the result of a given floating-point operation is correct to R significant digits. Thus, the table defines a lower bound on the precision to which the arguments must be computed to avoid loss of significance. Additional digits in the arguments will not increase the number of significant digits (in the result) that are later used.

TABLE 2. MINIMUM PRECISION NEEDED FOR DESIRED SIGNIFICANCE

Expression to be Evaluated	Precision Needed for X	Precision Needed for Y
X ± Y	R	R
X × Y	R	R
X / Y	2R	R
X < Y	R	R
X ← Y	-	R

A special case can arise for the X±Y case in Table 2. It is clear that R digits are required in the arguments and that R digits are sufficient for X+Y when X and Y have similar signs and for X-Y when X and Y have different signs. If, however, X±Y results in computing the difference between the absolute values of X and Y and the difference is small, the result will contain leading zeros. These are, however, significant zeros and do not require special consideration in Table 2. Because such leading zeros are significant, it is important that the hardware not raise the underflow interrupt when the result of an addition or subtraction cannot be normalized.

In practice, it is not always possible to compute as many significant digits as are specified for an expression. This can occur when operands have less precision (and, therefore, less significance) than is needed, either because they are variables of limited precision or because they are expressions that cannot be computed to the desired significance. If the argument is an expression in X and Y of precisions (and assumed significance) P and Q, respectively, then the number of significant digits in the computed value of the expression can be no greater than that shown in Table 3.

TABLE 3. MAXIMUM OBTAINABLE PRECISION WITHOUT INSIGNIFICANT DIGITS

Expression	Maximum Precision Obtainable
X ± Y	Max(P,Q)+1
X × Y	P+Q
X / Y	Min(P,Q)
X < Y	Min(P,Q)
X ← Y	Min(P,Q)

The number of digits that should be computed for an expression (and for which the user did not explicitly specify the precision) should be the minimum of the values as obtained from Tables 2 and 3. Any precision greater than that obtained from Table 2 represents computation of digits that will not be used. Any precision greater than that obtained from Table 3 represents computation of digits that have no significance. For example, applying the tables to X ← X×Y/Z where X, Y, and Z each have precision R, confirms that X×Y should produce a double precision product.

Several observations can be made from these tables. The primary factor in limiting the growth in needed precisions for expressions will be assignments. If an exact value (such as a fixed-point number or an integer) is used in a floating-point addition, subtraction, or multiplication, then the precision needed to preserve significance would be infinite and, therefore, the expression should be evaluated to the full precision required for the result. If the needed precision is unknown, as might be the case for relational expressions, the maximum obtainable significance must be computed.

Computing the minimum number of digits from Tables 2 and 3 as a default, when the precision of an expression or subexpression is not explicitly specified, means that the explicit specification for the precision of an expression or subexpression must be interpreted as the needed precision when determining the precision from Table 2 and must be interpreted as the obtainable significance when the expression is considered as an operand in Table 3. That is, specifying the precision of a subexpression (3-1Da) has the same effect on precision as does assigning the value of the expression to a variable of the specified precision and then using the variable in the subsequent computation.

If the obtainable precision exceeds the needed precision, it may be possible to save computation time or space by computing only the needed precision. If the needed precision exceeds the obtainable precision, the number of significant digits will be less than the computed precision. The user should be made aware of the latter situation, possibly by a warning from the translator.

To implement a floating-point system efficiently, the translator must use the actual precisions and radix that are available in the object-machine hardware (3-1Ea). With proper rounding rules in the hardware, the precision that is implemented can exceed the specified precision without additional loss of significance in computational results. Thus, precision specifications (i.e., the maximum precision needed) shall be interpreted as the minimum precision to be implemented in the object machine (3-1Db).

To prevent loss of significance from additional digits of an implementation, the object machine must round all computational results to the precision implemented by the hardware (3-1Dc). The rounded result should differ from the true value by less than one in the least-significant digit and the difference should be equal to one only when the remainder of the unrounded result is exactly 1/2 to the number of digits actually computed inside the arithmetic hardware. In the latter case, any rounding rule that evenly distributes the direction of rounding is acceptable. Probably the best choice (although none should be dictated by the language, given the differences among current machines) is rounding to the nearest even, because this will reduce the number of one bits and, therefore, the probability that rounding from 1/2 will occur in subsequent computations.

Most floating-point hardware offers at least two choices of precision (often called single and double). If the translator compiles programs that use minimal storage and computational resources, it must be capable of implementing mixed-precision computations. This, in turn, requires conversions between the implemented precisions during execution. Because most conversions between specified precisions will not correspond to precision conversions in the object code, it will be difficult for the user to predict where the implemented conversions will occur and, in any case, the locations will be machine-dependent. Explicit conversions between specified precisions would seldom generate machine code. Also, the user generally will not know whether (a) an assignment reduces the precision with an accompanying loss of information, (b) requires more significance than is available on the right-hand side, or (c) does not involve a change in precision. If explicit conversions are required, users will tend to use explicit conversion operations on every floating-point assignment and their presence will serve no useful purpose. Consequently, explicit conversion operations shall not be required between floating-point precisions (3-1Dd). Translators should, however, inform the user when information is lost or when the desired precision is unobtainable. Neither of these two situations is necessarily an error.

Because precision specifications are interpreted as the maximum needed and the minimum to be implemented, they need not be specified precisely. In particular, the precision could be specified in decimal digits, even though implemented on a binary machine with a slightly higher binary precision.

Although the derivation of the "best" precision for a given computation is complicated, the user need only know (a) that the implementation will produce the maximum obtainable significance within the specified precisions (where it is assumed that the number of significant digits in a variable is equal to the variable's specified precision), (b) that a warning will be generated during translation if the specified precision is greater than the obtainable significance, and (c) that the translator has the information necessary to minimize the time and space required during execution without losing significance. The translator writer need only (a) understand why mechanical application of Tables 2 and 3 will produce the least (and, therefore, least expensive) lower bound on the precision without loss of significance, and (b) know that, except for changes in precision across the hardware single/double precision boundary, the resulting object code will be unaffected by the precision calculations.

C. FLOATING-POINT OPERATIONS

Because floating-point values are finite approximations to real numbers, each floating-point number corresponds to an infinite set of real numbers. Floating-point arithmetic computations can only produce values that approximate the corresponding results using real arithmetic (thus, floating-point computations are said to be inexact). Although a floating-point number corresponds to infinitely many real values, it has only one value; that value will be called the designated value (i.e., the abstract value designated by the floating-point number). Floating-point operations can be defined only in terms of the values that can be represented (i.e., in terms of the designated values).

Four arithmetic operations - addition, subtraction, multiplication, and division - are needed in floating-point computations (3-1Bb). Absolute value is also useful and is difficult to implement efficiently in terms of the other four operations. Each floating-point arithmetic operation should be defined so that its result approximates as closely as possible the true value resulting from application of the corresponding mathematical operation (using real arithmetic) on the designated values of the operands. Rounding rules for floating-point computation were discussed in Section IV-B.

Although floating-point arithmetic is inexact, it can and should retain the properties of the corresponding real arithmetic functions wherever possible. It cannot be guaranteed that multiplication or division will distribute over addition or subtraction, that operations will be associative, or that values will have exact reciprocals. Addition and multiplication should, however, be commutative. Because the lack of associativity does not affect the results in many calculations, translators should be able to assume that floating-point operations are associative in order to produce the most efficient execution, except in those cases where the program contains explicit parentheses to designate the execution order desired (4Ga). The rules suggested in the previous section for defining the precision to be implemented will preserve commutativity, providing the corresponding hardware operations are commutative (this can be done, even though the rules may call for mixed-precision calculations).

There are three exception conditions that can arise in floating-point computations (10A). Detection of the error when an actual value exceeds the specified range is not normally provided by floating-point hardware and must be implemented in software with substantial execution cost. Consequently, it is likely that its use will often be suppressed (10B, 10G) by the user. Errors on the implemented range can occur in the primitive floating-point operations of the language and can be efficiently detected using exponent overflow hardware interrupts (10B). Errors in precision arise when a floating-point value cannot be normalized because the unnormalized representation already has the minimal (i.e., most negative) representable exponent. Thus, the actual precision is reduced by the number of leading zeroes. Detection of errors from loss of precision can be efficiently implemented using exponent-underflow hardware interrupts. It is important that the hardware raise the underflow interrupt when the result of a multiplication or division cannot be normalized, rather than when the result is zero (i.e., when some rather than all significant digits are lost).

Arithmetic operations such as integer division and remainder require exact arguments and are, therefore, inappropriate for floating-point arguments. When exact computations are to be applied to floating-point values, the floating-point values can be (explicitly) converted to fixed point and the appropriate fixed-point operation applied.

Relational operations are needed in all floating-point computations (3-1Bc). Because floating-point numbers are approximations to real numbers, a comparison between the designated values of floating-point numbers does not necessarily produce the same result as would comparison between the real values that are represented by the floating-point numbers. Nevertheless, floating-point relational operations should have a precise meaning that preserves the mathematical properties of the corresponding real operations. In particular, $< \leq = \geq >$ should be transitive, $=$ should be commutative and reflexive, and for any floating-point values X and Y, $X<Y$ iff $Y>X$, $X\leq Y$ iff $Y\geq X$, $X\neq Y$ iff $\neg(X=Y)$, $X\leq Y$ iff $X<Y$ or $X=Y$, and $X\geq Y$ iff $X>Y$ or $X=Y$.

These properties can be achieved easily by implementing floating-point relations as exact comparisons between the designated values. Notice, however, that these properties will not always be preserved in combination with floating-point arithmetic (e.g., $X < Y$ does not imply that $X+Z < Y+Z$).

Other useful arithmetic operations, including square root and trignometric functions, should be available as standard library definitions (3-1Bf, 12A). Particular important library functions are exponentiation to integer powers and logarithm, which are needed for conversion from symbolic representations to floating point on input and for conversion from floating point to symbolic scientific notation for display, respectively. If the library interface is sufficiently transparent to the user, the difference between built-in operations and standard library definitions should be indistinguishable to the user.

Because the actual implementation of a given floating-point computation will vary from machine to machine and because the numerical results are affected by the details of the implementation (although, if properly used, the number of significant digits will be identical), the language should provide operations that can be used to access the actual precision, radix, and exponent range used in the implementation of a variable or expression (3-1Eb). Because these properties of floating-point representations are fixed during translation, the corresponding functions can be treated as constants during translation.

To facilitate the writing of generic definitions, the language should provide a user function that can be evaluated during translation (12Da) to access the (explicit or implied) specified precision of a variable or expression. A precision operation would be most useful in generic definitions where it could be used to specify the precision of local floating-point variables. A precision operation could also be used to improve the readability of explicit conversions to floating point (e.g., FLOAT(X,PRECISION(Y)) meaning convert X to a floating-point number having the same (specified) precision as Y). Because the precision of an expression is bound during translation, the precision operation can always be evaluated during translation. Inclusion of a precision operation will permit the definition of standard mathematical functions without prior knowledge of the precision needed.

D. LITERALS AND FIXED-POINT VALUES IN FLOATING-POINT COMPUTATIONS

Fixed-point values are sometimes used as arguments in floating-point computations. In order to convert from fixed point to floating point, the precision needed for the result must be known. Any precision less than that needed to obtain the maximum precision that would otherwise be obtainable from the floating-point expression containing the fixed-point argument can unnecessarily reduce the significance of the result. Any precision greater than that needed to produce the maximum obtainable significance for the floating-point expression may introduce unnecessary execution cost. Thus, there is just one "best" precision and it can be determined from the context during translation. Although the precision could be determined by the user, the determination is nontrivial and error-prone if done by hand. In any case, it must be done by the translator to determine the most efficient implemented precision (independent of what precision is specified). Consequently, an explicit precision parameter is undesirable for conversion from fixed point to floating point.

As a general rule, explicit conversion operations should be required for conversions between types (3Ba). By this general rule and for consistency with other conversions between types, an explicit operator should be required for conversion from fixed point to floating point. The reasons which lead to the general rule, however, do not apply in this case. Although there is a change in the interpretation, it is a relaxation (exact to inexact) and, therefore, is not error-prone. Although there can be a change in the designated value and in the physical representation, conversion will be into that floating-point representation that corresponds to the abstract value of the fixed-point value (without introducing loss of significance in subsequent computations). That is, the abstract value will still be represented. An explicit conversion is not needed to designate result precision (and even if desired, could be accomplished by explicitly specifying the precision of the resulting floating-point expression). Explicit conversions from fixed point to floating point introduce extra notation in programs, thereby increasing the chance of error, without adding additional information and without adding useful redundancy that could be checked by the translator. If explicit operations are required for conversion from fixed point to floating, programs will be more difficult to write, read, and understand without other compensating advantages. The language should not require explicit conversion from fixed to floating point. Such implicit conversions will permit expressions to produce floating-point results from operands, some of which are fixed-point.

As was seen in Section IV-B, it is possible to correctly evaluate expressions that have results of limited precision from operands that, in some cases, are exact (i.e., that have infinite precision). For example, one might want to assign the product of the integer two and a floating-point value X to a variable Y of precision P. Although 2×X would have to be computed to infinite precision to avoid loss of information, in practice it need be computed only to P significant digits to obtain the same value in Y (within 1 in the last digit) as would be obtained by computing 2×X to infinite precision and then rounding to P digits.

There shall be built-in numeric literals (2Ga). Numeric literals are needed to designate numeric constants in programs. A literal is a symbolic representation of a constant value and in common usage designates some one abstract value (e.g., in decimal notation 61.2 is exactly one-tenth of the integer 612). The value of a literal can be represented exactly in a variety of fixed-point scales, but may not be exactly representable in the available floating-point representations (e.g., 61.2 is not exactly representable in any floating-point representation with exponents to base 2, 8, or 16).

Literals, therefore, must be converted to the nearest floating-point value of appropriate precision. As with fixed-point values that are used in floating-point computations, the "best" precision can be determined automatically during translation and should not be specified as an explicit parameter. It is not necessary to make floating-point literals syntactically distinct from fixed-point literals because whether a literal is fixed or floating point is easily determined from the context (i.e., floating-point operations have at least one nonliteral floating point argument). Thus, floating-point and fixed-point literals can share the same syntactic forms without complicating the language or its use.

REFERENCES

1. High Order Language Working Group, *Department of Defense Requirement for High-Order Computer Programming Languages --Revised IRONMAN*, July 1977

2. *"WOODENMAN" Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language*, Institute for Defense Analyses Working Paper, David A. Fisher, 13 August 1975

3. "The Common Programming Language Effort of the Department of Defense," 1977 Computers in Aerospace Conference, David A. Fisher

4. *DoD Higher Order Programming Language*, Memorandum issued by Malcolm R. Currie, Director, Defense Research and Engineering (DDR&E), 28 January 1975

5. High Order Language Working Group, *Department of Defense Requirements for High Order Computer Programming Languages --TINMAN*, June 1976

6. Institute for Defense Analyses, *A Common Programming Language for the Department of Defense--Background and Technical Requirements*, Paper P-1191, AD-A028297, D. A. Fisher, June 1976

7. *Lecture Notes in Computer Science, Vol 54, Design and Implementation of Programming Languages--Proceedings of a DoD Sponsored Workshop, October 1976*, Eds. John H. Williams and David A. Fisher, Springer-Verlag, 1977

8. *Language Evaluation Coordinating Committee Report to the High-Order Language Working Group (HOLWG)*, S. Amoroso, P. Wegner, D. Morris, D. White, AD-A037634, 14 January 1977 with appendices by:

   a. Lloyd Campbell, Army Ballistic Research Laboratory, Aberdeen, Maryland

   b. P. Parayre, Centre de Programmation de la Marine, Paris, France

c. J. D. Ichbiah, CII-Honeywell Bull, Louveciennes, France

d. Computer Sciences Corporation, Falls Church, Virginia

e. A. Demers and J. Williams, Cornell University, Ithaca, New York

f. Jean E. Sammet, Maurice Ackroyd, Michael L. Bell, I. Gray Kinnie, and Richard Kopp; IBM Federal Systems Division, Gaithersburg, Maryland

g. Brian L. Marks, and Robert F. Maddock, IBM United Kingdom Laboratories, Winchester, England; and Tom C. Spillman, IBM Federal Systems Division

h. J. G. P. Barnes, Imperial Chemical Industries Limited, Slough, England

i. B. M. Brosgol, R. E. Hartman, J. R. Nestor, M. S. Roth, and L. M. Weissman, Intermetrics, Inc., Cambridge, Massachusetts

j. Stephen L. Squires, National Security Agency, Ft. Meade, Maryland

k. Dr. Tomas Martin, PEARL Development Board, c/o Gesellschaft fur Kernforschung MBH, Karlsruhe, W. Germany

l. RLG Associates, Inc., Reston, Virginia

m. E. F. Miller and A. I. Wassermann, Science Applications, Inc., San Francisco, California

n. John B. Goodenough, Clement L. McGowan, and John R. Kelly, SofTech, Inc., Waltham, Massachusetts

o. Software Sciences Limited, Farnborough, Hampshire, England

p. Texas Instruments Incorporated, Huntsville, Alabama

9. High Order Language Working Group, *Department of Defense Requirements for Higher Order Computer Programming Languages-IRONMAN*, 14 January 1977

APPENDIX

EXCERPTS FROM *REVISED IRONMAN*

Paragraphs of the *Revised IRONMAN* relevant to the numeric processing facilities for the common language are:

Paragraph

1A through 1G
2G
3B
3-1A through 3-1H
3-3B
3-3G
4A
4B
4G
7G
8A
8B
10A
10B
10F
10G
11A
11C
12A
12D
13D
13F

1A. Generality. The language shall provide generality only to the extent necessary to satisfy the needs of embedded computer applications. Such applications require real time control, self diagnostics, input-output to nonstandard peripheral devices, parallel processing, numeric computation, and file processing. The language shall not contain features that are unnecessary to satisfy the requirements.

1B. Reliability. The language should aid the design and development of reliable programs. The language shall be designed to avoid error prone features and to maximize automatic detection of programming errors. The language shall require some redundant, but not duplicative, specifications in programs. Translators shall produce explanatory diagnostic and warning messages, but shall not attempt to correct programming errors.

1C. Maintainability. The language should promote ease of program maintenance. It should emphasize program readability over writability. That is, it should emphasize the clarity, understandability, and modifiability of programs over programming ease. The language should encourage user documentation of programs. It shall require explicit specification of programmer decisions and shall provide defaults only for instances where the default is stated in the language definition, is always meaningful, reflects common usage, and can be explicitly overridden.

1D. Efficiency. The language design should aid the production of efficient object programs. Constructs that have unexpectedly expensive or inexpensive implementations should be easily recognizable by translators and by users. Where possible, features should be chosen to have a simple and efficient implementation in many object machines, to avoid execution costs for available generality where it is not needed, to maximize the number of safe optimizations available to translators, and to ensure that unused and constant portions of programs will not add to execution costs. Execution time support packages of the language shall not be included in object code unless they are called.

1E. Simplicity. The language should not contain unnecessary complexity. It should have a consistent semantic structure that minimizes the number of underlying concepts. It should be as small as possible consistent with the needs of the intended applications. It should have few special cases and should be composed from features that are individually simple in their semantics. The language should have uniform syntactic conventions and should not provide several notations for the same concept.

1F. Implementability. The language shall be composed from features that are understood and can be implemented. The semantics of each feature should be sufficiently well specified and understandable that it will be possible to predict its interaction with other features. To the extent that it does not interfere with other requirements, the language shall facilitate the production of translators that are easy to implement and are efficient during translation. There shall be no language restrictions that are not enforceable by translators.

1G. Machine Independence. The language shall strive for machine independence. It shall not dictate the characteristics of object machines or operating systems. The design of the language shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system. There shall be a facility for specifying those portions of programs that are dependent on the object machine configuration and for conditionally compiling programs depending on the actual configuration.

2G. Numeric Literals. There shall be built-in numeric literals. Numeric literals shall have the same values in programs as in data.

3B. Implicit Type Conversions. There shall be no implicit conversions between types. Differences in range, precision, scale, and representation shall not be interpreted as differences in type.

3-1A. Numeric Values. The language shall provide an integer and fixed point type and a floating point type. Numeric operations and assignment that would cause the most significant digits of numeric values to be truncated (e.g., when overflow occurs) shall constitute an exception situation.

3-1B. Numeric Operations. There shall be built-in operations (i.e., functions) for conversion between numeric types. There shall be built-in operations for addition, subtraction, multiplication, division with floating point result, and negation for all numeric types. There shall be built-in equality (i.e., equal and unequal) and ordering operations (i.e., less than, greater than, less than or equal, and greater than or equal) between elements of each numeric type. Numeric values shall be equal if and only if they represent exactly the same abstract value. The semantics of all built-in numeric operations shall be included in the language definition. [Note that there might also be standard library definitions for numeric functions such as exponentiation.]

3-1C. Numeric Variables. The range of each numeric variable must be specified in programs and determinable at its allocation time. Such specifications shall be interpreted as the minimum range to be implemented. Explicit conversion operations shall not be required between numeric ranges.

3-1D. Floating Point Precision. The precision of each floating point variable and expression shall be specifiable in programs and shall be determinable at translation time. Precision specifications shall be required for each floating point variable. They shall be interpreted as the minimum precisions to be implemented in the object machine. Floating point results shall be implicitly rounded to the implemented precision. Explicit conversion operations shall not be required between floating point precisions.

3-1E. Floating Point Implementation. A floating point computation may be implemented using the actual precision, radix, and exponent range available in the object machine hardware. There shall be built-in operations to access the actual precision, radix, and exponent range with which floating point variables and expressions are implemented.

3-1F. Integer and Fixed Point Numbers. Integer and fixed point numbers shall be treated as exact numeric values. There shall be no implicit truncation or rounding in integer and fixed point computations.

3-1G. Fixed Point Scale. The scale or step size (i.e., the minimal representable difference between values) of each fixed point variable must be specified in programs and be determinable at translation time.

3-1H. Integer and Fixed Point Operations. There shall be built-in operations for integer and fixed point division with remainder and for conversion between fixed point scale factors. The language shall require explicit scale conversion operations whenever the abstract value may be changed.

3-3B. Component Specifications. For elements of composite types, the type of each component (i.e., field) must be explicitly specified in programs and determinable at translation time. Components may be of any type (including array and record types). Range, precision and scale specifications shall be required for each component of appropriate numeric types.

3-3G. Variant Types. It shall be possible to define types with alternative record structures (i.e., variants). The structure of each variant shall be determinable at translation time. The value of a variant may be used anywhere a value of the variant type is permitted (i.e., if A is a variant of B, then elements of type A may be used anywhere type B is allowed).

4A. Form of Expressions. The parsing of correct expressions shall not depend on the types of their operands or on whether the types of the operands are built into the language.

4B. Type of Expressions. The language shall require that the type of the value of each expression be determinable at translation time. It shall be possible to specify the type of an expression explicitly. [Note that this does not provide a mechanism for type conversion.]

4G. Effect of Parentheses. If present, explicit parentheses shall dictate the association of operands with operators. Explicit parentheses shall be required to resolve the operator-operand associations wherever an expression has a nonassociative operator to the left of an operator of the same precedence and wherever consecutive operators of an expression are of the same precedence but have different operand types.

7G. Parameter Specifications. The type of each formal parameter must be explicitly specified in programs and shall be determinable at translation time. Parameters may be of any type. Range, precision, and scale specifications shall be required for each formal parameter of appropriate numeric types. A translation time error shall be reported wherever corresponding formal and actual parameters are of different types and wherever a program attempts to use a constant or an expression where a variable is required.

8A. Low Level Input-Output Operations. There shall be a set of built-in low level input-output operations that act on physical files (e.g., input-output channels and peripheral devices). The low level operations shall be chosen to insure that all application level input-output operations can be defined within the language. They shall include operations to send control information, to receive control information, to begin transfer of data in either direction, and to wait for completion of a data transfer.

8B. Application Level Input-Output Operations. There shall be standard library definitions for application level input-output to logical files. These shall include operations for creating, deleting, opening, closing, reading, writing, positioning and formatting logical files. The meaning of such operations shall depend on the general characteristics of the files or devices (e.g., on whether they are sequentially or randomly accessed), but shall not be dependent on a specific device.

10A. Exception Handling Facility. There shall be an exception handling mechanism for responding to unplanned error situations detected during execution. The exception situations shall include errors detected by hardware, software errors detected during execution, error situations in built-in operations, and user defined exceptions. Exceptions should add to the execution time of programs only if they are enabled.

10B. Error Situations. The errors detectable during execution shall include exceeding the specified range of an array subscript, exceeding the specified range of a variable, exceeding the implemented range of a variable, attempting to access an uninitialized variable, dynamic aliasing of array components, attempting to access a field of a variant that is not present, termination of a parallel path, and failing to satisfy a program specified assertion. [Note that many of these checks can be done or partially done during translation, thereby reducing execution costs. Several are very expensive in execution unless aided by special hardware, and consequently will often be suppressed (see 10G).]

10F. Assertions. It shall be possible to include assertions in programs. If an assertion is false when encountered during execution, it shall enable an exception. Translators shall give warning if an assertion has side effects. [Note that assertions can also be used to aid optimization and maintenance.]

10G. Suppressing Exceptions. It shall be possible at translation time to suppress individually the detection of exceptions within a given scope. Should an exception situation occur when its detection is suppressed, the consequences will be unpredictable. An exception must not be enabled nor reenabled in a scope in which it is suppressed. [Note that suppression of an exception is not an assertion that the enabling error will not occur.]

11A. Data Representation. The language shall permit but not require programs to specify the physical representation of data. These specifications shall be distinct from the logical descriptions. Specifications for the order of fields, the width of fields, the presence of "don't care" fields, the positions of word boundaries, and the object representation of atomic data shall be allowed. If object representations are not specified, they shall be determined by the translator.

11C. Translation Time Constants and Functions. The translator shall require the specification of the object machine configuration including the machine model, the memory size, special hardware options, the operating system if present, and peripheral equipment. The translator shall use this specification when generating the object code. The language shall supply translation time constants and functions so that, during translation, programs can access the object machine characteristics and can check properties of the program components including their types, their specified and implemented ranges, their specified representation, whether an exception is suppressed, whether an actual parameter is a translation time constant, and the current optimization criteria.

12A. Library Entries. The language shall support the use of an external library. Library entries shall include type definitions, input-output packages, common pools of shared declarations, application oriented software packages, other separately compiled segments, and machine configuration specifications. The library shall be structured to allow entries to be associated with particular applications, projects, and users.

12D. Generic Definitions. It shall be possible to define functions, procedures and encapsulations that have generic parameters. Such parameters shall be instantiated during translation at each call and may be any defined identifier (including those for variables, functions, types, or representations), any expression, or any statement. Generic parameters shall be evaluated in the context of the call. [Note that a generic definition is a restricted form of macro, often cannot be separately compiled, and that where generic definitions are implemented as closed routines, several instantiations can often share the same object code.]

13D. Translator Diagnostics. Translators shall be responsible for reporting errors that are detectable at translation time and for optimizing object code. If it can be guaranteed that a function or procedure call will not terminate normally, then the exception shall be reported as a translation error at the point of call. Translators shall do full syntax and type checking, shall check that all language imposed restrictions are met, and shall provide warnings where constructs will be unusually expensive in execution. A recommended set of translation time diagnostic and warning messages shall be included in the language definition.

13F. Translation and Execution Restrictions. Translators should fail to compile correct programs only when the program exceeds the resources or capabilities of the intended object machine or when the program requires more resources during the translation than are available on the host machine. An error shall be reported when a program requires memory, devices, or special hardware that are unavailable in the object machine. Neither the language nor its translators shall impose arbitrary restrictions on language features. For example, they shall not impose restrictions on the number of array dimensions, on the number of identifiers, on the length of identifiers, or on the number of nested parentheses levels unless such restrictions are dictated by unavoidable limitations of the host machine. The size of object programs and data structures shall be limited only by the object machine characteristics. All such restrictions shall be documented in user accessible manuals.