JPL PUBLICATION 78-92

Proceedings

Conference on the Programming Environment for Development of Numerical Software

Cosponsored by
JPL and ACM-SIGNUM

Hilton Hotel
Pasadena, California
October 18-20, 1978

October 18, 1978

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology,
Pasadena, California

JPL PUBLICATION 78-92

Proceedings

Conference on the Programming Environment for Development of Numerical Software

Cosponsored by
JPL and ACM-SIGNUM

Hilton Hotel
Pasadena, California
October 18-20, 1978

October 18, 1978

National Aeronautics and
Space Administration

Jet Propulsion Laboratory
California Institute of Technology,
Pasadena, California

This publication was prepared by the Jet Propulsion Laboratory, California Institute of Technology, under NASA Contract NAS7-100

CONFERENCE OVERVIEW

C L. Lawson, Conference Chairman

THE NUMERICAL SOFTWARE COMMUNITY, especially in North America, has predominately used the Fortran language In the late 60's, there may have been thoughts that some other language might soon replace Fortran for scientific computing. By the early 70's, however, this appeared much less likely and a number of people began developing preprocessors and other software tools that would make some of the newer systematic approaches to software development and testing more readily available in Fortran environments.

The 1974 SIGNUM Workshop on Fortran Preprocessors for Numerical Software held at the Jet Propulsion Laboratory, Pasadena, spotlighted this developing area and initiated closer communication between SIGNUM and the ANSI Fortran Committee X3J3. Contacts established at the 1974 Workshop led to the organization of two Fortran Forum meetings in 1976, one in California and one in New York, providing early public discussion of the emerging Fortran 77 standard

The Workshop and the two Forums gave added impetus to a changing mood within X3J3. There was increased willingness to deal with significant additions to the language. In particular, it seems likely that the eventual decision to add the structured IF into Fortran 77 was strongly influenced by the considerable interest in structured Fortran evidenced at the Workshop.

For the present conference we have carried the SIGNUM-X3J3 communication a step further by arranging for a half-day joint meeting of the two groups Brainerd and Schenk will speak on the current directions of X3J3 and Lawson, Brown, and Smith will present some views from the mathematical software community for consideration by X3J3. An extended discussion period is scheduled for more general interaction between the SIGNUM conference attendees and the members of X3J3.

Boyle, Dritz, Schryer, Crary, and Presser will describe a variety of software tools applicable to the development of numerical software. Rice gives an analysis of the increased programming efficiency attainable by use of a very high order problem oriented language for solving PDE's Miller presents a case for an EISPACK-type research and development project in the area of software tools. Osterweil speaks on strategies for making the best use of software tools.

The organizations with the most pressing necessity to make systematic and efficient use of tools for dealing with mathematical software are the vendors of commercial mathematical libraries. We will hear from Fox of the PORT library, Aird of IMSL, and du Croz of NAG on their methods and experiences

Approaches to the technical problems associated with the exchange of mathematical software between different facilities will be treated in talks by Butler and Van Snyder.

New preprocessor-based languages will be presented by Grosse and Feldman. PASCAL, which has been regarded mainly as a teaching language has achieved a new burst of popularity among microcomputer users, due in large part to PASCAL compiler developments at UCSD. Volper of that group will discuss PASCAL's relevance to the mathematical software community.

The DoD High Order Language Working Group is in the midst of one of the largest efforts ever undertaken by a user organization to create and promote the use of an entirely new language The talk by Fisher on this project will give conference attendees an opportunity to begin to assess the significance of this project for numerical software.

Hull will discuss language and hardware features that would give the numerical analyst new levels of control and confidence in numerical computations.

Kahan and Eggers are both presently active on an IEEE committee to specify a standard for floating-point arithmetic hardware. They will present two different proposals on this subject.

Fong will report on the recent DoE Workshop on Mathematical Libraries Jones will report on the DoE Advanced Computing Committee Language Working Group which is developing guidelines for Fortran extensions to meet DoE needs.

It is a pleasure to thank Tom Aird, Jim Boyle, and Norm Schryer for working with me in organizing the technical program for this conference John Rice provided valuable assistance in coordination with ACM Headquarters, publicity, and budgetary guidance.

The Jet Propulsion Laboratory has supported the conference by permitting me and my secretary, Kay Cleland, to carry through the preparation, and by publishing the conference schedule and proceedings for distribution at the conference I wish to personally thank Kay for her sustained dedication to the success of the conference.

The members of the ANSI Fortran Committee X3J3 will be making a very generous effort on behalf of the conference in travelling thirty miles across Los Angeles from their own meeting site to Pasadena for the Wednesday morning sessions I thank Jeanne Adams, Chairman of X3J3, and James Matheny, local arrangements Chairman fo X3J3's Oct 16-19 meeting in Los Angeles for their unstinting cooperation in arranging for X3J3's participation in this conference

I wish to express appreciation to all of the speakers and their home institutions for their outlay of time, funds, and energy to bring their recent work to the podium of this conference. The Session Chairmen and attendees at the conference encompass a wide range of experience in mathematical software and we look forward to stimulating exchanges of information during the discussion periods. We believe this conference will provide a useful snapshot of the current state-of-the-art in tools and techniques for development of mathematical software and we hope this will lead to more widespread use of systematic methods in this field.

CONTENTS

CONTENTS (cont.)

N79-12716

A "CORE + MODULES" APPROACH TO FORTRAN STANDARDIZATION

Walt Brainerd
Los Alamos Scientific Laboratory
Los Alamos, NM 87544

ABSTRACT

The ANSI FORTRAN standards committee X3J3 has adopted a "core + modules" approach to specifying the next revision of the standard. The motivation for and the projected benefits of this approach are discussed.

In 1978 January, ANSI X3J3 voted to adopt a framework consisting of a "core" and "modules" for developing the next revision of the ANSI Fortran standard. Of course, this is a decision which could be reversed if the approach appears to be unsuitable or technically unsound after some experimentation. However, the approval of this procedure is an indication that the committee wants to invest considerable effort in an attempt to make this approach work.

There are at least three reasons for adopting the "core + modules" approach:

1) to provide a mechanism to interface with collateral standards and implementations in major applications areas
2) to provide a mechanism for having optional functional areas described within the standard
3) to specify a smaller, more elegant, language than Fortran 77 without decreasing the status of Fortran 77 as a standard language.

Each of these reasons is now discussed in more detail.

One of the major concerns of X3J3 is the development of collateral standards in areas such as data base management, real time process control, and graphics. X3J3 does not have the resources to do the technical development of standards in all of these areas; in some cases X3J3 may not even be involved directly in the approval of such a standard. Therefore, it is important that X3J3 provide a scheme whereby collateral standards in these applications areas can be regarded as modules in the language that are "attached" to the core of the language in a standard way. The only mechanism considered so far for interfacing with these modules is through the CALL statement, extended to allow the arguments to be identified by key words. This topic is covered in more detail in another paper and is an area that can use more good ideas, because it is a very difficult and important problem.

A second kind of extension might be called a "language feature module." This sort of module would include a collection of related language features that might not be appropriate to include in the core, but which should have its form specified so that all extensions to the core in this area will be the same. Example candidates for such modules are array processing, a bit data type, and specification of numerical precision. Fortran 77 should be considered to be such a module.

It may be quite inappropriate to add some of these language features to Fortran 77. For example, it would be rather messy to add a bit data type or REAL*11 (indicating at least 11 digits of precision) on top of the Fortran 77 equivalencing mechanism.

For these reasons it is important to design a core that is sufficiently trim that new language features can be added in a natural way.

Since Fortran 77 will be one of the modules, the core need not be constrained to contain all archaic features of Fortran. One of the design objectives is to eliminate those features (e.g., the arithmetic IF statement) that are no longer necessary, due to the addition of better equivalent features or those features (e.g., storage association) that actually stand in the way of adding features recognized as contributing to high quality programming practices.

To provide just one example illustrating how the storage association concept impedes the addition of useful features, consider the possibility of a conditional array assignment.

```
REAL A(90), B(90), C(90), D(90)
A(*) = 0
B(*) = 0
WHERE (A(*) .LT. 2) DO
      C(*) = B(*) + 1
      END WHERE
```

If no equivalencing is allowed, the assignment may be implemented as

```
      DO 9 I = 1, 90
    9 IF (A(I) .LT. 2) C(I) = B(I) + 1
```

However, if the program may contain the statement

```
EQUIVALENCE (C(1), B(2))
```

the loop above will set C(I) = I for I = 1 to 90 instead of setting each element to 1. The implementation will be more complex on most machines.

In 1978 August, X3J3 approved a proposal to create a first cut at a core language by starting with Fortran 77 and making the following changes. Of course, this list is not final, but is given to provide a flavor of the final result. When reading the list of changes, it is important to keep in mind that Fortran 77 will be one of the modules, so any compiler that contains the Fortran 77 module will be able to process programs containing any of the features of Fortran 77.

The following two paragraphs are excerpted from the X3J3 proposal to indicate some of the objectives of the approach.

The general philosophy governing this core design is that the core should be comprehensive, containing virtually all of the generally useful features of Fortran and that it should form a practical, general-purpose programming language. Modules would be used largely for special-purpose language features that entail high implementation costs or are used primarily in special-purpose application areas. The number of such modules should remain small in order to minimize problems of program portability. Three examples might be (1) a module providing comprehensive array processing facilities, (2) one providing data base management facilities, and (3) one providing features of Fortran 77, and possibly certain other isolated special-purpose features, not contained in the core.

Another goal is to produce a more elegant language by moving redundant features and including features which lend themselves to modern programming practices.

The net effect of these changes is the following:

1) Subroutine linkage facilities are enhanced to improve the interface with applications modules written in Fortran.
2) Archaic control structures are replaced with modern ones.
3) The concept of storage association is removed.
4) Fixed-form source is replaced with free-form source.

There are two kinds of changes: features added to Fortran 77 and features remaining in Fortran 77 but not included in the core.

To be added	To be moved to Fortran 77 module
Free-form source	Column 6 continuation
Larger character set	C for comment
Longer names	
Simple data structures	EQUIVALENCE
Some array processing	COMMON and BLOCK DATA
Global data definition	Passing an array element or substring to a dummy array
Bit data type	Association of ENTRY names
A length (digits) for REAL	DOUBLE PRECISION
Enhanced looping	Arithmetic IF
Case construct	Computed GO TO
Internal procedures	Alternate RETURN
Subroutine linkage	ASSIGN and assigned GO TO
	Statement functions
	ERR = and END = specifiers
	H, X, and D edit descriptors
	Specific names for intrinsics

D2

NUMERICAL PRECISION AND DATA STRUCTURES

Werner Schenk, Xerox Corporation
Rochester, New York

ABSTRACT

Group T9 of the ANSI Fortran Committee X3J3 has been assigned to study the areas of numerical precision, storage and data structures, with a goal of developing technical proposals and recommendations for future revisions of the Fortran standard. Developers and users of numerical software have proposed the addition of various functions to return the base of a computer system's number representation. Also desirable are features to enhance Fortran portability, such as single and double precision word lengths, and exponent ranges. Structured types proposed include arrays, records, sets and files.

INTRODUCTION

Soon after completing work on Fortran 77, the X3J3 Committee began the task of identifying technical issues to be considered for future revisions of the Fortran standard. As a first approach, the Committee reviewed all comments which had been received during the public review period of the proposed standard (Fortran 77), especially all of the comments which had been referred to the "future development" subcommittee.

This rough list of desirable features was sorted and categorized, resulting in the current X3J3 Committee organization of technical subgroups, such as T9, to investigate and propose specific changes. For the past year, group T9 has been gathering information about the issues of Numerical Precision and Data Structures.

NUMERICAL PRECISION FEATURES

L. D. Fosdick (1) * addressed the X3J3 Committee at their October 1977 meeting proposing a set of environment parameters and intrinsic functions to enhance computations and portability of numerical software. Since that time, group T9 has received proposals from W. S. Brown and S. I. Feldman (2) H. P. Zeiger (3), G. M. Bauer (4) outlining similar sets of desirable features. Additionally, group T9 has reviewed the discussions of B. Ford (5) (6) on transportable numerical software.

Fosdick's proposed features address portability, reliability, and efficiency of the Fortran language, with an automatic adjustment to changes in the environment. Examples cited are those of IMSL of Houston, Texas, and NAG of Oxford, England, who adjust their mathematical software libraries to a specific environment by removing coded records which do not apply to the environment from the source file. Environmental parameters identified include the following:

---

* Numbers in parenthesis designate References at end of paper.

1. Base of floating point representation.
2. Largest positive real number, exponent and integer.
3. Largest negative real number, exponent and integer.
4. Number of significant digits.
5. Exponent Bias.

Fosdick suggests that these parameters would be made available to a Fortran program with a set of intrinsic functions proposed by IFIP WG2.5 (EPSLN (a), INTXP (a), SETXP ($a_1 a_2$)).

Ford proposes three sets of parameters to make software transportable and adaptable to an environment. The arithmetic set, including the radix (same as Fosdick's base of a floating point representation); the Input/Output set, defining such entities as the standard units for input and output; and a miscellaneous set, to define number of characters per word, page size, and number of decimal digits.

Brown and Feldman present a language-independent proposal for environment parameters and basic functions for floating-point computation with specific representation in terms of generic functions. Their basic parameters include the base, the precision, the minimum exponent, and the maximum exponent. To provide access to precise and efficient numerical computation tools, Brown and Feldman suggest analysis and synthesis functions, such as exponent (x) and fraction (x); as well as precision functions to aid in iterational computations. Seven generic and six computational procedures are suggested, with specific illustrations of implementation for Univac 1100, Honeywell 6000, and Interdata 8/32 systems.

Group T9 expects to solicit additional proposals in the area of numerical precison from users and designers of numerical software. It is clear, that the above authors agree on a certain set of basic necessary parameters, although a wide range of nomenclature and specific function names have been proposed. Within the next year, group T9 will incorporate these suggestions into position papers and specific Fortran language proposals. The intent is to maintain close liaison with groups such as IFIP WG 2.5 to assure compatibility with language development and numerical computation practices.

DATA STRUCTURES

At the August 1978 meeting of X3J3, M. Freeman conducted a tutorial on "Data Structures: A Language Comparison." Feedback from committee members yielded the need for a glossary and definition of terms in this area. A survey questionnaire has been designed and will be mailed to X3J3 participants to reach some consensus as to the types of structures which should be con-

sidered for future Fortran revisions.

R. Oldehoeft and R. Page (7) have examined PASCAL data types as a possible model for Fortran. These types include arrays, records, sets and files. B. Lampson and others (8) describe the programming language EUCLID with its structures, arrays, records, and modules. In addition to defining the types of structures to be included in Fortran, group T9 will develop and propose necessary functions to operate on structures. Currently identified operations include: Read, Write, Assign, Initialize, and Compare. The array processing features are a separate issue being studied by group T6 of X3J3.

SUMMARY

Group T9 is nearing the conclusion of a period of information gathering. The next phase of technical work will be the identification of common features which have been proposed for numerical computation and data structures, followed by the development of Fortran language proposals for X3J3 committee consideration.

ACKNOWLEDGEMENT

The following members of X3J3 deserve credit for their participation in the technical work of group T9: L. Campbell, D. Rose, M. Freeman, L. Miller, and R. Page. In addition, Mr. Edward Nedimala of Xerox Corporation, provided valuable assistance in reviewing the various technical reports.

REFERENCES

1. L. D. Fosdick, "On Knowing the arithmetic environment in FORTRAN," A talk presented in conjunction with the X3J3 meeting, October 3, 1977.

2. W. S. Brown and S. I. Feldman, "Environment Parameters and Basic Functions for Floating-Point Computation," Bell Laboratories, Murray Hill, New Jersey.

3. H. P. Zeiger, Department of Computer Science, University of Colorado at Boulder. Letter to W. Schenk and X3J3, August 1, 1978.

4. G. M. Bauer, "Functions to Manipulate Floating-Point Data," Letter to X3J3, Group T9, August 29, 1978.

5. B. Ford et al, "Three Proposed Amendments to the Draft Proposed ANS Fortran Standard," FOR-WORD Fortran Development Newsletter, Vol. 2, No. 4, October 1976.

6. B. Ford editor, "Parameterization of the Environment for Transportable Numerical Software," SIGPLAN Notices, Vol. 13, No. 7, July 1978.

7. R. Oldehoeft and R. Page, "PASCAL data types and data structures," a report presented to X3J3 at Jackson, Wyoming, August 2, 1978.

8. B. W. Lampson et al, "Report on the Programming Language Euclid," SIGPLAN Notices, Vol. 12, No. 2, February 1977.

D3

N79-12718

THE IFIP WG 2.5 PROPOSALS ON FORTRAN

Charles L. Lawson

Jet Propulsion Laboratory
Pasadena, California

ABSTRACT

This paper is one of three to be presented at a joint meeting of SIGNUM and the ANSI X3J3 Fortran Committee October 18, 1978, for the purpose of communicating suggestions arising in the mathematical software community to the X3J3 Committee. A summary is given of language problem areas and possible solutions that have been discussed by the IFIP Working Group 2.5 on Numerical Software. Also included are some thoughts on control structures due to the author.

THE MATHEMATICAL SOFTWARE COMMUNITY has given serious and continuing attention to portability and black-box modularity. This is evidenced by the existance of extensive and widely used libraries and other systematized collections of portable or transportable mathematical subprograms which are ready for use in applications without need for modification or recompilation.

We feel this approach has had large payoffs in injecting good quality subroutines into innumerable applications programs. There is a great deal of room for improvement, however, and some of this improvement could be facilitated by enhancements in programming languages. Following are three general concepts that are often among the goals in designing and programming mathematical software and which could be better handled with the aid of appropriate language enhancements.

1. Long argument lists are to be avoided. The user should not be burdened with declaring, dimensioning, and assigning variables that are inessential to his or her functional concept of the task to be done by the subprogram.

2 Portability. Ideally it should be possible to move the source code to different machines with no changes. Next best is code that can be ported by systematic changes effected by a processor designed for that purpose. Since a library may involve hundreds of subprograms, manual changes that would be acceptable in porting an individual subprogram do not provide a reliable approach.

3. A library subprogram should be useable without the need to make changes dependent on the application. The user has enough concerns without also being asked to reset dimensions in a set of library subprograms and recompile them.

TOPICS IN LANGUAGE ENHANCEMENT CONSIDERED BY WG2.5

The topics listed in this section have been discussed by WG2.5 [see Appendix A for information on WG2.5]. Some of these have only been briefly considered by WG2.5 while others have been the subject of a substantial amount of effort including the solicitation of comments via the SIGNUM Newsletter and the formulation of detailed proposals for language enhancements. I believe it is fair to say that the mathematical software community, including WG2.5, is more concerned that some viable solutions be worked out for these problem areas than that any particular suggested solution be adopted.

DOUBLE COMPLEX. There are significant classes of problems in engineering and science for which complex numbers provide the most natural and convenient mode of expression. If it is agreed that a programming language for scientific computation must support complex arithmetic, then it should also provide a choice of precisions for complex arithmetic for the same reasons that such a choice is provided for real arithmetic. See Ref [1].

ARRAYS OF WORK SPACE IN LIBRARY SUBROUTINES. Many subroutines in a mathematical library require one or more arrays of temporary work space of sizes depending on the problem parameters. Suppose for example, a library subroutine SUB requires two temporary integer arrays of length N and one of length M where N and M are dummy arguments of SUB. How should this work space be provided to SUB?

One possibility is to include the three arrays as distinct dummy arguments. This is objectionable as it leads to long argument lists which, among other things, can discourage a potential user considering the use of SUB. When libraries are modularized so that SUB may call lower level library subroutines which in turn call others, etc., this approach can lead to some very long argument lists since temporary arrays needed only by some lower level subprogram will appear in the argument lists of all higher level subprograms.

Use of COMMON storage as specified in Fortran 66 and 77 is not suitable since the lengths of arrays in COMMON in a set of precompiled library subprograms cannot adjust to the problem variables M and N.

A frequently used approach is to require one array of length 2*N+M as an argument to SUB and then make use of this array as two arrays of length N and one of length M. WG2.5 has proposed in some detail a "MAP" statement, Ref [1], to provide dynamic renaming of subsets of dummy arrays to

facilitate this approach. This capability is also supported efficiently by the "DEFINE" statement in Univac Fortran V.

Another approach to this problem could be through changes to the concept of COMMON. For example, there could be two types of COMMON declaration, primary and secondary. A primary declaration establishes absolute amounts of storage needed just as the present COMMON declaration does whereas a secondary declaration may contain array names with adjustable dimensions. Each distinct named COMMON block would have to be declared by a primary declaration in at least one program unit of a complete executable program but could also be declared by secondary declarations in other program units.

This concept appears to be convenient and efficient for sets of library subprograms that are always used together. It may not be convenient for a library subprogram that is sometimes called by other library subprograms and sometimes directly by users.

Yet another approach would be truely dynamic arrays that a black-box subroutine could fetch from and release to the operating system as needed.

CALLS FROM A LIBRARY SUBPROGRAM TO USER CODE. Certain mathematical subprograms require access to user-provided problem-dependent code. For example, a subprogram for numerical integration (quadrature) requires access to the user's code for evaluation of the integrand. Analagous situations arise with library subprograms for differential equations, nonlinear equations, optimization, sparse matrix algorithms, etc.

The approach supported by Fortran and most other languages would be typified by a user main program MAIN calling a library subprogram, say QUAD, which in turn calls a user-coded function evaluation subprogram FUNC.

It is not uncommon that a need arises for a direct data path between MAIN to FUNC. For example, one may need to integrate a number of functions, in which case quantities which are input or computed by MAIN need to be communicated to FUNC to select or parameterize the function to be computed.

In Fortran this can be handled by inserting a COMMON block in MAIN and FUNC. The user is then required to write a number of things twice - the COMMON statements and possibly some type and dimension statements. This double writing and maintenance could be alleviated, however, if an INCLUDE feature were available.

An alternative approach sometimes called "reverse communication" has been used in some library software. In this approach there is no separate FUNC subprogram. The code that would have been in FUNC is in MAIN. When QUAD needs a function value instead of calling FUNC it returns to MAIN with a branching index set to cause MAIN to branch to the function evaluation code and then call QUAD again. In this approach the user needs to write and maintain only one program unit, MAIN, instead of two, MAIN and FUNC. Furthermore, there is no need for COMMON to establish a data path between the main driver and the function evaluation code.

Reverse communication can be confusing to a user and may involve more linkage overhead since it will involve just one call to QUAD and many calls to FUNC. Generally, FUNC would be a simpler code than QUAD in ways that might cause its linkage overhead to be less.

If internal procedures are added to Fortran, as for instance was done in Univac Fortran V, then FUNC could be written as an internal procedure within MAIN and one could have the more easily understood structure of QUAD calling FUNC and still have only one program unit for the user to write and maintain, and have data accessible to the driver and FUNC. To support this usage it must be possible to pass the name of the internal procedure FUNC to QUAD in the call from MAIN so that QUAD can call FUNC.

DECOMPOSITION AND SYNTHESIS OF FLOATING-POINT NUMBERS. There are situations in which it is desirable to decompose a floating-point number into its exponent and mantissa parts or conversely to construct a floating-point number from these two parts. One application would be the scaling of a vector by an exact power of the machine's radix in order to avoid introducing rounding errors in the scaling operation. Another application would be in writing portable subprograms for certain special functions such as SQRT, EXP, and LOG.

Let b denote the radix of the system used to represent floating-point numbers on a particular computer. Then each nonzero number x, representable on this computer, can be associated uniquely with a pair of numbers (e,f) by the requirement that

$x=fb^e$, $b^{-1}\le|f|<1$, and e is an integer.

WG2.5 has suggested the following two functions

1. Integer exponent: INTXP(x)

   This function returns the integer value e if $x\neq0$ and x is associated with the pair (e,f). The result is undefined if x=0.

2. Set exponent: SETXP(x,N)

   If x=0 the result is zero. If $x\neq0$ and x is associated with the pair (e,f) then the result is the value fb^N if this value does not underflow or overflow, otherwise the result is undefined.

As examples of usage, suppose $x\neq0$ and x is associated with the pair (e,f). Then e and f can be obtained by:

```
INTEGER e
e = INTXP(x)
f = SETXP(x,0)
```

and x can be constructed from e and f by

```
x = SETXP(f,e)
```

The machine radix b can be obtained by

```
b = SETXP(1.0,2)
```

It is further proposed that these functions be generic to deal with both single and double precision floating-point numbers.

ENVIRONMENT PARAMETERS. One of the key hinderances to portability in Fortran has been the need to include machine dependent parameters in programs. For example, after strenuous effort to achieve portability in the EISPACK eigenvalue codes, it remained that a number representing the arithmetic precision needed to be reset in certain subroutines to adjust the code to different machines.

A quite thorough discussion of this problem with reasonable approaches to its resolution was given

by Redish and Ward Ref [2] in 1971. Redish and Ward noted that the problem had previously been discussed by Naur Ref [3] in 1964

The approach to defining and naming the set of environment parameters used in the IMSL library was reported in Aird et al Ref [4]. This paper stimulated an ad-hoc meeting on the subject by T. Aird, G. Byrne, B. Ford, and F. Krogh during a SIGNUM conference in Pasadena, Nov 9, 1974 Ref [5]. The draft produced by this ad-hoc meeting was used as a working paper at the first meeting of WG2.5 in January, 1975, and after further exposure and discussion, particularly at the Oak Brook Portability meeting, June 1976, evolved to the paper approved by WG2.5 and the parent IFIP Committee TC2 and published in Ref [6] and elsewhere.

This paper proposed definitions and names for three clases of quantities as follows:

Arithmetic Set
: Radix, mantissa length, relative precision, overflow threshold, underflow threshold, and symmetric range.

Input-Output Set
: Standard input unit, standard output unit, standard error message unit, number of characters per standard input record, and number of characters per standard output record.

Miscellaneous Set
: Number of characters per word, size of a "page" in a virtual storage system, and number of decimal digits useful to the compiler in numeric constants.

PRECISION FUNCTION. WG2.5 has proposed a function to be used to determine the resolution of a computer's number system in the vicinity of a given floating-point number. The set P of all real numbers that can be represented as valid floating-point numbers of a particular precision, e.g., single precision, in storage in a given computer forms a linearly ordered set. Let σ denote the underflow threshold, i.e , the smallest positive real number such that both σ and $-\sigma$ are members of P. The proposed precision function is EPSLN(x) whose value is $\max\{x-x', x''-x, \sigma\}$ where

$$x' = \begin{cases} \text{The predecessor of x in P if there is one.} \\ \text{x if x is the least member of P.} \end{cases}$$

and

$$x'' = \begin{cases} \text{The successor of x in P if there is one.} \\ \text{x if x is the greatest member of P.} \end{cases}$$

OPTIONAL ARGUMENTS FOR SUBPROGRAMS. For some mathematical subprograms it is desirable to provide the userwith the choice of a simple calling sequence with a minimal number of parameters or a longer calling sequence giving the user more detailed control of the subprogram This is presently accomplished by providing different "front-end" subprograms or by "option-vectors" which are arrays in the calling sequence the elements of which essentially play the role of arguments in a variable length or keyworded calling sequence.

The concept of keyworded calling sequences currently being considered by X3J3 may be a very useful mechanism for this situation.

VIEWS OF THE AUTHOR ON CONTROL STRUCTURES

For the purpose of this discussion "structured programming" will be defined to mean programming using some specified set of control structures designed to encourage and support the production of programs whose control logic has a high degree of human comprehensibility. The author's opinions on structured programming are strongly influenced by his use of JPL's structured Fortran preprocessor SFTRAN, Ref [7], over the past three years.

Since human comprehension of a program listing is a major goal of structured programming, the format of a listing and the suitability of the control structures for being formatted in a rational way are important issues.

```
* * * * * * * * * * * * * * * * * * * * * * * * *
*                                                 *
* Suggestion 1: Each control structure should     *
* have explicit beginning and ending statements.  *
* * * * * * * * * * * * * * * * * * * * * * * * *
```

For example, the IF(p)THEN and ENDIF play these roles for the structured IF of Fortran 77 and of SFTRAN. In contrast, there is no explicit ending statement for the IF of ALGOL 60 or PASCAL and this leads to the necessity of special nonintuitive rules to match ELSE's with the correct IF's in cases of nested IF's. Furthermore, the lack of an explicit ending statement for the WHILE in PASCAL, and the presence of one (the UNTIL) for the REPEAT, leads to the peculularity that the segment of code controlled by the WHILE is a simple statement or a BEGIN-END block, whereas the code controlled by the REPEAT is a sequence of statements.

The presence of explicit beginning and ending statements also suggests a natural rule for indenting listings: The explicit beginning and ending statements of the same structure are listed at the same indentation level. Lines between these two statements are further indented with the exception of certain secondary keyword lines, such as ELSE, that are listed at the same level as the associated beginning and ending statements.

```
* * * * * * * * * * * * * * * * * * * * * * *
*                                             *
* Suggestion 2: A compiler for structured     *
* Fortran should be required to produce a     *
* cononically indented listing.               *
* * * * * * * * * * * * * * * * * * * * * * *
```

The problem of a syntax for special exits from structures has appeared in minutes of recent X3J3 meetings. The troublesome case of a special exit from a looping structure that is to skip past some code that follows the exited structure can be represented as an exit from an enclosing structure. In order to be able to place an enclosing structure where it is needed to serve this purpose, it is useful to have an essentially null structure such as DO BLOCK ... END BLOCK.

As an example consider:

```
DO BLOCK
  DO FOR I = N1,N2
     f
     IF(p) EXIT BLOCK
     g
  END FOR
  h
END BLOCK
```

It seems very likely that a programmer having a general acquaintance with control structures would correctly guess the control flow of this example, especially with the indentation as shown, which should always be supplied by the compiler.

I do not feel that this quality of self-evident semantics is shared by alternative approaches that have appeared in recent X3J3 minutes and use a single, but more complicated, control structure to express what has been represented in this example by a nesting of two elementary control structures.

An unconditional looping structure permitting one or more exits from anywhere in the loop is desirable. With such a structure one could do without special looping structures for the cases of testing at the beginning or testing at the end, however, I think these special cases occur so frequently that special structures should be provided for them.

When keywords are selected, I hope real words will be used rather than reversed words as in ALGOL68.

ACKNOWLEDGEMENTS

The account in this paper of language problem areas discussed by WG2.5 was greatly aided by WG2.5 working papers written by J. Reid and WG2.5 minutes written by S. Hague and B. Ford. The present paper is solely the responsibility of the author. It has not been reviewed before publication by other members of WG2.5 and nothing in it should be quoted solely on the authority of this paper as being the view of WG2.5.

Other speakers at this SIGNUM meeting, W. S. Brown, B. T. Smith, and T. E. Hull, will also be presenting material that has been discussed by WG2.5.

This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract No. NAS 7-100, sponsored by the National Aeronautics and Space Administration.

REFERENCES

1. Ford, B., Reid, J., and Smith, B., Three Proposed Amendments to the Draft Proposed ANS FORTRAN Standard, SIGNUM Newsletter, 12, No. 1, (1977), pp 18-20.

2. Redish, K. A., and Ward, W., Environment Enquiries for Numerical Analysis, SIGNUM Newsletter, 6, No. 1, (1971), pp 10-15.

3. Naur, P., Proposals for a new language, Algol Bulletin, 18, (Oct. 1964), pp 26-31.

4. Aird, T. J., et al, Name Standardization and Value Specification for Machine Dependent Constants, SIGNUM Newsletter, 9, No. 4, (1974), pp 11-13.

5. Editorial, SIGNUM Newsletter, 10, No. 1, (1975), p 2.

6. Ford, B., Parameterization for the Environment for Transportable Numerical Software, ACM TOMS, 4, No. 2, June, 1978, pp 100-103.

7. Flynn, J. A , and Carr, G. P., User's Guide to SFTRAN II, JPL Internal Document 1846-79, Rev A, October, 1976.

APPENDIX A

The IFIP Working Group 2.5 on Numerical Software currently consists of sixteen people as follows: E. L. Battiste, W. S. Brown, L. D. Fosdick, C. W. Gear, C. L. Lawson, J. C. T. Pool, J. R. Rice, and B. T. Smith of the U.S.A., and T. J. Dekker, The Netherlands; B. Einarsson, Sweden; B. Ford, U.K.; T. E. Hull, Canada; J. K. Reid, U.K.; C. H. Reinsch, West Germany, H. J. Stetter, Austria, and N. N. Yanenko, U.S.S.R. The group has met at approximately one year intervals since its initial meeting in January, 1975.

Environment Parameters and Basic Functions for Floating-Point Computation

W S Brown and S I Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

This paper presents a language-independent proposal for environment parameters and basic functions for floating-point computation, and suggests a specific representation in terms of generic functions for Fortran 77. The environment parameters were originally introduced in 1967 by Forsythe and Moler [1], who attributed the essentials of their theory to Wilkinson [2]. These parameters are also used in the PORT mathematical subroutine library [3], with precise definitions in terms of a more recent model of floating-point computation [4], and a similar set has been proposed by the IFIP Working on Numerical Software [5] Three of the basic functions are taken from a proposal by Ford, Reid, and Smith [6], but redefined in terms of the parameters and the model to provide a firm theoretical foundation. The other three basic functions can be expressed in terms of these, but we feel they should be provided separately for convenience

The stated purpose of the model is to capture the fundamental concepts of floating-point computation in a small set of parameters and a small set of axioms In this paper we extend the earlier work by proposing basic functions to analyze, synthesize, and scale floating-point numbers, and to provide sharp measures of roundoff error.

Using the proposed parameters and functions, one can write portable and robust codes that deal intimately with the floating-point representation Subject to underflow and overflow constraints, one can scale a number by a power of the floating-point radix inexpensively and without loss of precision. Similarly, one can take an approximate logarithm of a floating-point number very cheaply by extracting the exponent field, and one can readily implement algorithms (e.g., those for logarithmic, exponential, and n^{th} root functions) that operate separately on the exponent and fraction-part. The convergence of iterations is extremely important in numerical computation While one often wants to relate the termination conditions to the accuracy of the host computer, it is essential to avoid demanding more accuracy than the computer can provide Although a termination criterion can be formulated in terms of the environment parameters alone, it may be desirable to use the roundoff-measuring functions for finer control, especially when the floating-point radix is greater than 2.

We view it as essential to provide mechanisms for accomplishing these goals in any language that is used for scientific computing Ideally, to facilitate translations from one language to another, these mechanisms ought to be provided in a similar manner in all such languages Therefore, we present our proposal in a language-independent form, before suggesting a specific representation for Fortran

2. Environment Parameters

In this section we present the environment parameters of the model, and review other key properties First, for any given real number $x \neq 0$, we define the (integer) *exponent, e,* and the (real) *fraction-part, f,* relative to a specified (integer) *base* $b \geqslant 2$, so that

$$x = fb^e$$
$$b^{-1} \leqslant |f| < 1. \tag{1}$$

Next, we introduce the parameters of the model — four basic integer parameters and three derived real parameters, all constants for a given floating-point number system. If a computer supports two or more such systems (e.g., single- and double-precision), then each has its own parameters. The basic parameters are

1. The *base*, $b \geqslant 2$.
2. The *precision*, $p \geqslant 2$
3. The *minimum exponent*, $e_{\min} < 0$.
4. The *maximum exponent*, $e_{\max} > 0$.

These must be chosen so that zero and all numbers with exponents in the range

$$e_{\min} \leqslant e \leqslant e_{\max} \tag{2}$$

and fraction-parts of the form

$$f = \pm(f_1 b^{-1} + \quad \cdot + f_p b^{-p})$$
$$f_1 = 1, \ldots, b-1$$
$$f_i = 0, \ldots, b-1, \qquad i = 2, \ldots, p \tag{3}$$

are possible values for floating-point variables. These *model numbers* are a subset of all the *machine numbers* that can occur in floating-point computation.

Returning to (1), it is evident that the model numbers with a given exponent e are equally spaced, a change in f of one unit in the last place implies a change in x of b^{e-p}. It follows that the *maximum relative spacing* is

$$\epsilon = b^{1-p} \tag{4}$$

Also of interest are the smallest positive model number

$$\sigma = b^{e_{\min}-1}, \tag{5}$$

and the largest model number

$$\lambda = b^{e_{\max}}(1-b^{-p}). \tag{6}$$

From the point of view of the model, the integer parameters b, p, $e_{\min}$, and $e_{\max}$ are fundamental, a practical programmer is more likely to want the real parameters σ, λ, and ϵ

3. Analysis and Synthesis Functions

As noted in Section 1, it is often necessary in numerical computation to scale a number by a power of the base, to break a number into its exponent and fraction-part, or to synthesize a number from these constituents. To provide convenient access to precise and efficient versions of these operations, we propose the following functions:

exponent(x) returns the exponent of x, represented as an integer; if $x=0$, the result is undefined

fraction (x) returns the fraction-part of x; if $x=0$, the result is 0

synthesize (x,e) returns *fraction* (x) b^e if possible, otherwise the result is undefined

scale (x,e) returns xb^e if possible, otherwise the result is undefined However, if $0<|xb^e|<\sigma$, then any numerical result must be in the interval $[0,\sigma]$ if $x > 0$ or in the interval $[-\sigma,0]$ if $x < 0$

4. Precision Functions

To attain sharp control over the termination of an iteration, one needs to know the absolute or relative spacing of model numbers in the vicinity of a given number x If $x = fb^e$, we have already shown (see Section 2) that the absolute spacing is b^{e-p}, and it follows that the relative spacing is $b^{-p}/|f|$ Unfortunately, if $|x| < \sigma/\epsilon = b^{e_{\min}+p-2}$, then the absolute spacing is less than σ, and hence too small to be represented in the model. This suggests defining the *absolute-spacing* function

$$\alpha(x) = \begin{cases} b^{e-p}, & \text{if } |x| \geqslant \sigma/\epsilon \\ \sigma, & \text{if } |x| < \sigma/\epsilon \end{cases} \tag{7}$$

and the *relative-spacing* function

$$\rho(x) = \begin{cases} b^{-p}/|f|, & \text{if } x \neq 0, \\ \text{undefined}, & \text{if } x=0 \end{cases} \tag{8}$$

Instead of including $\rho(x)$ in the basic set, we favor the *reciprocal-relative-spacing* function

$$\beta(x) = 1/\rho(x) = |f|b^p, \tag{9}$$

because its definition is simpler, its evaluation is faster and involves no roundoff, and it is more often wanted.

5. Implementability

Each of the seven environment parameters is a well defined constant for any given floating-point number system. Although it may be convenient to express these parameters as functions (see Section 6), the compiler should substitute the correct values rather than producing code to fetch them at run-time

Each of the six basic functions is simple enough to permit a short in-line implementation on most machines. Furthermore, the definitions are meaningful for all real x, except that *exponent* (0) is undefined Finally, each function can be evaluated without error whether or not x is a model number, provided only that the result is representable; however, if x is initially in an extra-long register, it may be rounded or chopped before the computation begins.

6. Fortran Representation

In all of the above, we have carefully ignored the distinction between single- and double-precision numbers. The Standard Fortran language specifically has floating-point variables of these two precisions; some compilers recognize a third. There is talk of adding a mechanism to Fortran to permit specifying the number of digits of accuracy, rather than the number of machine words To avoid difficulties in this area, we propose using generic functions, for which the compiler chooses the operation to be performed and/or the type of the result from the type of the first argument Like the conversion functions in Fortran 77, the proposed functions need not have specific names for the different types. The only restriction on such generic functions is that they cannot be passed as actual arguments.

The following seven generic functions (in which the prefix "EP" stands for "Environment Parameter") would provide the necessary parameters

$$\begin{aligned}
\text{EPBASE(X)} &= b \\
\text{EPPREC(X)} &= p \\
\text{EPEMIN(X)} &= e_{\min} \\
\text{EPEMAX(X)} &= e_{\max} \\
\text{EPTINY(X)} &= \sigma \\
\text{EPHUGE(X)} &= \lambda \\
\text{EPMRSP(X)} &= \epsilon
\end{aligned}$$

The first four of these functions return integers related to the precision of X. The last three return floating-point values with the same precision as X. The functions *EPBASE, EPPREC,* and *EPHUGE* should also be defined for integer arguments; an appropriate model of integer computation is outlined in [3]

For the six computational procedures, we suggest

$$\begin{aligned}
\text{FPABSP(X)} &= \alpha(X) \\
\text{FPRRSP(X)} &= \beta(X) \\
\text{FPEXPN(X)} &= \textit{exponent}(X) \\
\text{FPFRAC(X)} &= \textit{fraction}(X) \\
\text{FPMAKE(X,E)} &= \textit{fraction}(X)b^E \\
\text{FPSCAL(X,E)} &= Xb^E
\end{aligned}$$

where the prefix "FP" stands for "Floating Point". FPEXPN returns the (integer) exponent of X, the other five functions return floating-point values with the same precision as X

7. Acknowledgments

We thank R H. Bothur for his assistance with the Univac codes in the appendix We also thank the participants in stimulating discussions at the Los Alamos Portability Workshop (Los Alamos, New Mexico, May 1976), the Argonne National Laboratory Workshop on the Portability of Numerical Software (Oak Brook, Illinois, June 1976), the Mathematics Research Center Symposium on Mathematical Software (Madison, Wisconsin, March 1977), and two meetings of the IFIP Working Group on Numerical Software Finally, we are indebted to numerous colleagues at Bell Laboratories for helpful comments on several earlier drafts

References

1. G. E Forsythe and C B Moler, *Computer Solution of Linear Algebraic Systems,* p 87, Prentice-Hall, Englewood Cliffs, N.J., 1967.
2. J. H. Wilkinson, *Rounding Errors in Algebraic Processes,* Prentice-Hall, Englewood Cliffs, N J , 1963.
3. P. A. Fox, A. D Hall, and N L Schryer, "The PORT Mathematical Subroutine Library," *ACM Transactions on Mathematical Software 4,* pp 104-126, June 1978
4. W S Brown, "A Realistic Model of Floating-Point Computation," *Mathematical Software III,* ed John R Rice, pp 343-360, Academic Press, New York, 1977
5. Brian Ford, "Parametrisation of the Environment for Transportable Numerical Software," *ACM Transactions on Mathematical Software 4,* pp 100-103, June 1978.
6. B Ford, J. K Reid, and B T. Smith, "Three Proposed Amendments to the Draft Proposed ANS Fortran Standard," *ACM SIGNUM Newsletter 12,* pp 18-20, March 1977.

D5

A Comparison of Two Recent Approaches to Machine Parameterization for Mathematical Software

N79-12720

Brian T. Smith*, Applied Mathematics Division
Argonne National Laboratory
Argonne, Illinois, U.S A.

INTRODUCTION

Recently, there have been published two different proposals for expressing the dependence of numerical software on the environment. One proposal, which is essentially described in two papers [1,2] ** characterizes the dependence of the software on the machine architecture essentially in terms of the representation of floating point entities in the machine. The second, more recent proposal, described in [3] characterizes the dependence of the software on the environment in terms of models both of the floating point numbers and of the behavior of the arithmetic unit of the machine.

The purpose of this paper is to expose and clarify the differences and similarities between these two approaches. It is the author's opinion that the two approaches interpreted in their strictest sense serve two distinct purposes. When each is interpreted imprecisely, their roles overlap.

In order that the comparisons briefly discussed below are fully understood, we need to briefly summarize the separate approaches Since our interest here is in the parameterization of the numerical aspects of the environment, the non-numerical parameters discussed in [1] and the considerations of implementation discussed in [2,3] will not be considered here

PARAMETERIZATION IN TERMS OF THE REPRESENTATION

In [1], there is described and defined a collection of parameters which characterizes the set of all floating point numbers in terms of their representations on current machines. The assumption underlying this characterization, which is not explicitly stated, is that all machines use a positional notation with a fixed radix and fixed length to represent floating point entities Thus, quantities such as the radix of the representation, the number of radix digits in the significand, the smallest and largest positive numbers in the set of floating point numbers (denoted "representable" numbers) such that both the number and its negation are representable, are defined. These parameters, then, characterize the salient properties of the representable floating point numbers.

Other parameters are included which have more relevance to preparing mathematical software. For example, the relative precision parameter, known in various papers as MACHEP, is defined so that it can be used to determine negligible numbers compared to one in additive operations. Another such parameter is the symmetric range parameter which is the largest positive representable number such that its negation, its reciprocal, and its negative reciprocal each can be approximated by a representable number within a relative error of the relative precision parameter.

At first sight, the latter parameters seem to be simply related to the former set, however, such relations are not machine independent in general.

To complement this set of parameters, three functions were defined which permitted access to the size and representation of any floating point number. Briefly, one function determined the integer exponent of an entity X, when expressed in a standard exponent-fraction form, another determined a number that was small relative to X (that is, the largest positive number X which could be considered negligible when involved with X in additive operations), and a third function formed a floating point entity from an integer exponent and another entity containing the fractional part,

The above definitions were designed to make the parameterization of the environment dependent as much as possible on the representation of floating point entities. This is not strictly adhered to in terms of the detailed definitions in at least two instances. The relative precision parameter and the negligible number function are defined in terms of the additive operations and so depend on the arithmetic unit This anomoly could be avoided by defining these items in terms of the number of radix digits in the representation. The second instance where a definition was not tied to the representation of entities is that the integer exponent function is defined in terms of a canonical exponent-fraction representation of numbers instead of the representation of the machine. This was done for uniformity of the returned result on different machines for the same numbers and at the same time permits floating point numbers to be decomposed and synthesized without the introduction of rounding errors

One goal of the above approach was to define the parameters and functions basically in terms of the representation. As the above discussion illustrates, such definitions may conflict with other important considerations such as portability, and here the definition of the exponent-fraction manipulation functions was modified to satisfy the more important consideration of portability.

This goal may also conflict with the desire to obtain suitable and precise definitions for all hardware. A case in point is the definition of the relative precision parameter where the definition given in [1] breaks down on unusual hardware. This parameter is defined in [1] as the smallest number e such that for the computed and stored quantities 1-e and 1+e, $1-e<1<1+e$. This definition is not

*Work performed under the auspices of the U.S Department of Energy.
**Numbers in brackets designate references at the end of the abstract.

suitable for a machine with a large number of guard digits whose rounding strategy is to round to the nearest odd significand. The resulting e for such a machine is not suitable for the relative precision parameter as it cannot be used to measure negligible numbers in additive operations with numbers near 1 as well as 1 itself. One other common definition for e is the radix raised to a power of 1 minus the number of radix digits in the representation but this is not completely satisfactory for processors that perform proper rounding.

Consequently, we see a major disadvantage with this approach in constraining the definitions of the parameter and functions to the representation of floating point numbers. It appears to be very difficult to define the parameters in a portable yet reliable manner for all machines. For this approach to be workable as an implemented feature in a language, the definitions may need to be adjusted to each environment to satisfy the intent of the original definitions

These difficulties with this approach lead one naturally to the second approach [2]. Rather than treating directly the diverse computing environments, it might be better to define a model of a computer, and then state the definitions of the parameters and functions in terms of the model.

THE MODEL APPROACH

Recently in [2], Brown et al describe a characterization of the environment in terms of a parameterized model of the floating point number system and arithmetic unit. The model characterizes the representation of floating point numbers in a signed magnitude notation in terms of 4 parameters, and specifies the behavior of the arithmetic unit in terms of a small number of axioms. Numbers representable within the model are called model numbers; the model numbers for a specific machine may be a proper subset of the machine representable numbers. The parameters which characterize a particular environment are defined in terms of the specific values of the 4 parameters which determine the particular model. The environment functions which manipulate floating point entities are also defined in terms of the specific values of the 4 parameters.

The four parameters of the general model are: 1) the radix of the model numbers, 2) the effective number of base radix digits; 3) the maximum exponent for model numbers, and 4) the minimum exponent for model numbers. The environment parameters include these four parameters plus three others; 1) a large positive number near the overflow threshold, set equal to the radix raised to the power of the maximum exponent minus 1;2) a small positive number near the underflow threshold, set equal to the radix raised to the minimum exponent, and 3) a number considered negligible when compared to 1, set equal to the radix raised to the effective number of base radix digits in the model minus 1.

The three basic analysis and synthesis functions are: 1) a function to extract the exponent of a floating point entity interpreted as a number within the model, 2) a function to extract the fraction of a floating point entity interpreted as a number within the model, and 3) a function to form a floating point entity from an integer exponent and the fractional part of a given floating point entity. Also, two basic precision functions are defined: 1) a function to determine the maximum absolute spacing in the model near a given floating point entity, 2) a function to determine the reciprocal of the maximum relative spacing in the model near a given floating point entity.

The key to understanding the approach is the specification of the effective number of base radix digits. The choice of this parameter is determined by the behaviour of the arithmetic unit. The idea is to penalize the specific model of the machine by reducing this number until a specified set of axioms and conditions characterizing the behavior of the arithmetic unit are all satisfied.

This approach now has three major advantages over the earlier approach. First, the definitions of the environment parameters are in terms of the general model and so can provide clean unambiguous definitions. Second, the intended use of the parameters can be specified clearly in terms of the model. And third, statements that specify the behavior of the software in terms of the model can conceivably be proven by relying on the axioms characterizing the model's arithmetic.

But it is just these axioms that make the model approach very difficult to use in practice. The difficulty comes in determining the effective number of radix digits. To be sure of your choice, one must carefully and thoroughly analyze the algorithms which implement the arithmetic operations on a specific machine. With straightforward arithmetic units, such verification is tedious but possible. With the more unusual arithmetic units, such verification can be very difficult indeed.

USES OF EACH APPROACH

We have referred to some of the advantages and disadvantages of each approach in terms of the ease with which the parameters and functions are defined In this section, we compare the uses of each approach.

In the first approach, the intent of the definitions is to constrain the parameters and functions to be dependent on the representation of the floating point numbers alone. None of the parameters (except for convenience and ease of definition) depend critically upon the behavior of the arithmetic unit. Consequently, the characterization of the machine environment using the first approach is most appropriate where the dependence of the software on the environment is in terms of the representation.

The second approach, on the other hand, applies to situations where the dependence of the software on the environment involves the behavior of the arithmetic unit. For example, algorithms that depend upon the size of rounding errors in each operation can thus be written in terms of the model parameters, thereby yielding reliable portable software Also, as the model guarantees a regular and controllable behavior for the arithmetic operations as specified by the axioms, and the precision functions as well, algorithms can more readily be analyzed and may be proven correct within the model.

Because of the manner in which the effective number of base radix digits is determined, the model is deteriorated by the least accurate arithmetic operation. Thus, a specific program which does not use such imprecise arithmetic operations may be unduly penalized.

Whereas the parameters and functions for the first approach are determined in general by the representation alone, some of the functions defined in the second approach are determined by both the model and the representation. For example, the function (x) returns the fraction determined by the model but is as precise as the machine; that is, the returned fraction may not be the fraction of any model number. The maximum absolute spacing function returns a number that is determined by the model alone and not the representation of the argument The maximum relative spacing, on the other hand, may return a number that is not a model number. Consequently, the algorithms that use the precision functions must be analyzed in terms of the possibly less precise model rather than the precision of the machine, despite the fact that the precision functions seem to address the representation of floating point entities on the machine.

CONCLUSION

Upon considering the two approaches to parameterization of the environment for floating point computation, there emerges two distinct uses for environment parameters. On one hand, the parameterization permits machine wide analysis of the algorithms, and on the other hand, permits machine wide control and formulation of algorithms for numerical software.

In the past, we have performed the analysis under the assumption that our algorithms would be executed on a well-behaved hypothetical arithmetic unit that satisfied some straightforward and useful axioms for floating point arithmetic. When implementing such algorithms, we had two choices, either machine constants were suitably adjusted where the constants were critical for the reliable behavior of the software so that the resulting software was safe, or machine constants were used directly where there was no danger of producing incorrect or misleading results.

Ideally, we are striving for a machine environment that makes this final choice unnecessary. Such an ideal requires the general availability of the perfect hardware. However, it is not clear that the perfect hardware is forthcoming in the near future. Thus, it seems inappropriate at this time to implement one parameterization of the environment to the exclusion of the other. Possibly, the two approaches can be merged so that we can have the best of both approaches.

REFERENCES

Ford, B., "Parameterization of the Environment for Transportable Software," ACM TOMS 4, June 1978.

Ford, B., Reid, J.K., Smith, B.T., "Three Proposed Amendments to the Draft Proposed ANS Fortran Standard," ACM SIGNUM Newsletter 12, pp. 18–20 March, 1977.

Brown, W.S., Feldman, S I., "Environment Parameters and Basic Functions for Floating-Point Computation," June 1978, private communication, also presented at ACM/SIGNUM Conference on The Programming Environment for Development of Numerical Software

D6

UNIX as an Environment for Producing Numerical Software

N. L. Schryer

Bell Laboratories
Murray Hill, New Jersey 07974

1. Introduction

The UNIX* operating system [1] supports a number of software tools which, when viewed as a whole, are an unusually powerful aid to programming.

The design, implementation, documentation and maintenance of a portable FORTRAN test of the floating-point arithmetic unit of a computer is used to illustrate these tools at work The result of this effort was a program created automatically from the paper describing the test. Thus, only the documentation had to be written, the program was automatically produced from it Also, changes in the document (debugging) were automatically reflected in the program.

The second section briefly describes the UNIX tools to be used The third section outlines the basic problem and the fourth section shows how the tools can help solve the problem.

2. Tools

This section lists and briefly describes the UNIX tools to be used

EQN - A mathematical equation-setting language [2].

When you want to say a_{ij}^{k} you simply type a sub ij sup k A general rule of thumb for EQN is that you type at it the words you would use in describing the object to a friend on the telephone The output of EQN is TROFF

TROFF- A phototypesetting language [1,pp2115-2135]

This processor lays out text according to user given commands and built-in rules. For example, the subheading for this paragraph was produced by typing

```
.SH
TROFF-
A phototypesetting language
```

where the SH command tells TROFF to underline the following input lines. TROFF also left and right justifies the text on the page, does hyphenation, and generally produces a tidy document from free-form input This paper is an example of its output.

EFL - A FORTRAN pre-processor language [4].

This pre-processor has an Algol-like input syntax, portable FORTRAN [3] as output, and provides a language of considerable power and elegance It has the usual control-flow constructions (IF ELSE . ., WHILE, FOR, etc.), as well as data structures and a macro facility. A useful feature of EFL is the ability to take input while inside one program file from another file during compilation, via the INCLUDE statement.

MAKE [5]

A UNIX command which makes sure that if A and B are two files, and B can be derived from A by a command sequence, then that command sequence is executed if and only if the date of A is later than the date of B Thus, MAKE is often used to keep object libraries up to

*UNIX is a trademark of Bell Laboratories.

date with respect to their source code

ED - The UNIX text editor [1, pp2115-2135]

For example, the ED command

g/b sup [^]*/s/b sup \([^]*\)/B(\1)/g

(don't worry, its easier to type than to read one of these!) changes all occurrences of b sup String into B(String), where String is any string of non-blank characters

SHELL - The UNIX command interpreter [1, pp1971-1990]

Each process on UNIX has a standard input and a standard output These standard i/o "devices" may be files, teletypes, or even other processes Thus, for example, the editor ED may take its editing commands from a file (script) Also, the output from one process may be input directly to another process This connection is called a "pipe" and is denoted by a "|". A typical use of a pipe is to create a document with the aid of EQN and TROFF, as in

EQN files | TROFF

where EQN produces TROFF input which is then shipped directly to TROFF to make the document.

3. The Problem

As part of the installation of the PORT library [6] it is necessary that the PORT machine parameterization be dynamically correct That is, it is not enough to simply read the owners manual for the host machine and conclude that it has a base-2 floating-point architecture with 48 bits in the mantissa The manner in which the the floating-point arithmetic units operate on their data must also be taken into account. For example, if the result of $a+b$ is only good to 24 bits in some cases, many algorithms aren't going to behave well if they believe that a rounding error is 2^{-48}

In order to test the accuracy of the floating-point arithmetic unit of a computer, we chose to compute

$$x \; op \; y$$

where x and y are one of

$$b^e \; (\; b^{-1} + b^{-t} \;) \qquad 1)$$

$$b^e \sum_{j=1}^{t} b^{-j} \qquad 2)$$

$$0 \qquad 3)$$

$$b^e \; (b-1) \sum_{j=1}^{t} b^{-j} \qquad 4)$$

$$b^e \; (b-1) \; (\; b^{-1} + b^{-t} \;) \qquad 5)$$

and op is any of $+, -, *$ and $/$ The test basically consists of finding the analytically correct value of $x \; op \; y$ and comparing it to what the machine gets for $fl \; (x \; op \; y)$

The fly in this ointment is that the exact result of $x \; op \; y$ must be found Take, for example, the product of two elements of pattern 1), denoted as 1)*1) We desire a $b^E M$ representation for the result The exponent of the result is trivial to compute The mantissa of the result can be rather simply computed as in

$$(b^{-1}+b^{-t1}) * (b^{-1}+b^{-t2}) = b^{-1}(b^{-1}+b^{-t1}+b^{-t2}+b^{-(t1+t2-1)})$$

This may be put into normalized form as follows

If (b=2 & $i1$=2 & $i2$=$i1$)
{ $b^0(b^{-1}+b^{-4})$ }
Else # No piling up
{ $b^{-1}(b^{-1}+b^{-i1}+b^{-i2}+b^{-(i1+i2-1)})$}

This is a rather simple example since most x *op* y derivations run from 6 to 8 TROFF output pages, but it does illustrate the technique. The problem here is that there are 42 separate x *op* y cases to be resolved, none of which is particularly complex individually, but which taken together represent a major effort - more than 200 TROFF output pages.

There is a grand ideal to which paper writers aspire - Document what you want to do, and then **do** it! Believing this, the author wrote the paper before writing any code A typist entered the text into a file, transcribing mathematical formulas into the notation of EQN, using the editor ED. As an example, the preceding display for the result of 1)*1) was entered as

```
If $(b=2~&~i1=2~&~i2=i1)$
        {       $b sup 0 (b sup -1 + b sup -4 )$         }
Else # No piling up
        {       $b sup -1 ( b sup -1 + b sup -i1
                + b sup -i2 + b sup -(i1+i2-1) )$       }
```

The "$" is a delimiter telling EQN what to act on, and the "~" tells EQN to leave a little white space

The problem now consists of implementing such formulae in a programming language Also, great care must be taken that the code agree with the document describing it This means that debugging such code (and formulae) must result in both the program and documentation being changed correctly and simultaneously. Yet, there are more than 200 pages of such formulae to implement!

4. The Solution

To attempt this by hand would be cosmic (and comic) folly. Soon neither the document nor the code would be correct, or in agreement with the other Actually, the author learned this the hard way, but lets not dwell on dead-ends. The solution is quite simple Use an ED script to convert the TROFF input into EFL and use MAKE to keep the whole thing up to date.

It is quite clear that the TROFF input for 1)*1) given earlier rather resembles an EFL program in structure (IF ... ELSE .), but not in detail - indeed, it is a rare language that can make sense of $b^{-1}(1+b^{-i})$! However, $b^{-1}(1+b^{-i})$ can be converted into a form EFL can recognize - B(1)*(1+B(i)) - by a rather general ED script fragment

```
g/b sup [^ ]*/s/b sup -\([^ ]*\)/B(\1)/g
g/) *(/s/) *(/)*(/g
```

and we can easily construct an array B such that $B(i) = b^{-i}$. A complete ED script may be constructed along the above lines It is a long (6 pages) but simple script The ED script applied to the TROFF input for 1)*1) gives the EFL program fragment

```
If (b == 2 & i1 == 2 & i2 == i1)
 {
 E = 0, M = (B(1)+B(HiLo(4)))
 }
Else   # No piling up
 {
 E =-1; M = ( B(1)+B(i1)+B(i2)+B(HiLo(i1+i2-1)))
 }
```

where HiLo is the statement function

HiLo(i) = Max(0,Min(i,t+1))

used to get the left-hand end-point of the smallest floating-point interval containing the exact result Here t is the number of base-b digits carried in the floating-point representation and the array B has been extended to have $B(0) = 0 = B(t+1)$. There are 42 such EFL program fragments They form the heart of the floating-point test, and the part with all the bugs in it - initially, at least! There is a standard EFL driver into which these fragments fit, via the EFL INCLUDE mechanism. The resulting 42 programs form the floating-point test

The above ED script mechanism produces the EFL code directly and automatically from the TROFF input Thus, only the TROFF input must be altered by hand, the EFL production is automatic Debugging was literally carried out at the TROFF (not the EFL) level

However, one great problem still remained The EFL depends on the TROFF input for 1)*1) How can one be sure that both the EFL and the document for 1)*1) have been produced from the most recent version of the TROFF input for 1)*1)? In all there are 42 such dependencies which must be checked Here MAKE is invaluable. A file is created for MAKE, giving the dependencies and desired command sequences Whenever the MAKE file is executed (by saying simply "make"), any TROFF input which has been altered since the last MAKE will be re-TROFFed, and any EFL file which has not been updated since its corresponding TROFF file was altered, will be updated and listed

Bibliography

[1] "UNIX Time-Sharing System", *BSTJ* **57**, 1897-2312(1978)

[2] B W Kernighan and L L Cherry, "A System for Typesetting Mathematics", *Comm. ACM* **18**, 151-157(1975).

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

[4] S I. Feldman, "EFL, an Extended FORTRAN Language", in preparation

[5] S I Feldman, "Make - A Program for Maintaining Computer Programs", Bell Laboratories Computing Science Technical Report #57, 1977

[6] P A. Fox, A.D Hall and N L Schryer, "The PORT Library Mathematical Subroutine Library", *TOMS*, **4**, 104-126(1978)

THE AUGMENT PRECOMPILER AS A TOOL FOR THE DEVELOPMENT OF SPECIAL PURPOSE ARITHMETIC PACKAGES

F. D Crary, The Boeing Company
Seattle, Washington 98124
J. M Yohe, Mathematics Research Center
University of Wisconsin - Madison
Madison, Wisconsin 53706

ABSTRACT

We discuss the use of a FORTRAN precompiler in the development of packages for nonstandard arithmetics In particular, the use of the FORTRAN precompiler, AUGMENT, renders the source code more lucid, reduces the number of lines of code in a nonstandard arithmetic package, facilitates modification, and ameliorates the problems of transporting such a package to another host system

1 INTRODUCTION

With decreasing hardware costs and increasing processor speeds, the cost of software development is becoming more and more a function of personnel cost. Furthermore, with the explosion of applications of digital computers, an ever-higher percentage of users place implicit trust in the software they use to support their applications.

For these reasons, it is essential to supply the user with reliable, well-documented software packages It is no longer profitable, or even feasible in many cases, to re-invent support software.

These considerations have led to an increasing emphasis on transportable software If development costs can be incurred just once for a package or system that will work correctly and accurately on a broad spectrum of equipment, users are willing to tolerate a reasonable amount of inefficiency in return for the convenience of having the development work done for them and the confidence that they can place in a quality product.

Increasingly, it is becoming practical to build on existing software rather than to develop new packages from first principles, even when the existing software might not be just exactly tailored to the application in question.

In order to make the best use of existing software, one must have the tools to make its incorporation in new programs reasonably easy, and one must adopt a design philosophy which will make the use of both the tools and the existing software natural and uncomplicated.

In this paper, we describe one such tool -- the AUGMENT precompiler for FORTRAN ([3]) -- and illustrate a design philosophy which has proved to be a reasonable application of the above criteria. Briefly, we advocate the abstraction of the data type representations to the maximum possible degree in the design and implementation of software packages, and subsequent application of the AUGMENT precompiler to bind the data type representations and extend them throughout the package.

We illustrate this philosophy with examples drawn from the interval arithmetic and triplex arithmetic packages developed by the second author

We also give an indication of several other applications of AUGMENT which, while not necessarily employing this philosophy, serve to indicate the breadth of possible applications of AUGMENT.

Sponsored by the U. S. Army under Contract No. DAAG29-75-C-0024.

2. BRIEF DESCRIPTION OF AUGMENT

AUGMENT is a program which allows the easy and natural use of nonstandard data types in Fortran. With only a couple of exceptions, it places nonstandard types on the same basis as standard types and allows the user to concentrate on his application rather than on the details of the data type implementation.

AUGMENT gains its power and ease of use through several aspects of its design.

(1) Its input language is very much like FORTRAN. The only changes are the addition of new type names and operators, and the ability to define "functions", naming parts ("fields") of variables, which may appear on either side of the assignment operator.

(2) AUGMENT is extremely portable Since it is written in FORTRAN, AUGMENT can be implemented on almost any computer. The machine-dependencies of AUGMENT are concentrated in eight subroutines which can be implemented in less than 200 lines of (machine-dependent) FORTRAN.

(3) AUGMENT's output is standard FORTRAN which makes it suitable as a cross-precompiler, that is, the AUGMENT translation may be performed on one (large) machine and the results compiled on or for some other machine which is unable to host AUGMENT.

There are three major steps in the use of AUGMENT.

Specification. The whole process begins with the specification of the properties of a nonstandard type. The specification will need to consider the following questions:

What information will the user see?
What operations will be made available?
How will this type interact with other types?

In many cases, the answers to these questions will be available in previous research or obvious from the nature of the new type. In other cases, con-

siderable research may be needed and even an an appeal to personal preference may be made

AUGMENT gives little assistance to this part of the process. The specifications will be guided by the applications envisioned by the person preparing the new type, by the operations known or felt to useful in manipulating the type, and aesthetic considerations such as consistency with similar types (if any) already existing in Fortran or previous extensions.

Binding (Implementation). The binding of the abstract specification of the new type to a representation usable through AUGMENT is by means of a "supporting package" of subroutines and functions, and through a "description deck" which tells AUGMENT about the supporting package. In this effort, the implementor must consider the conventions expected by AUGMENT in terms of argument number and order.

In addition to this, there may remain basic questions of representation. For example, the data structure which the user sees may not necessarily be the best way to implement the type.

Application. The application of AUGMENT to preparation of a program which uses one or more nonstandard data types is by far the easiest part of the process. Given that the supporting package(s), description deck(s), and adequate documentation have already been prepared, the use of the package(s) through AUGMENT consists of just four steps:

(1) Write the program using the new operators and functions.
(2) Supply AUGMENT with your program and the description deck(s)
(3) Compile AUGMENT's output with the system FORTRAN compiler.
(4) Link-edit and run.

3. ABSTRACT DATA TYPES

In the planning of most computations, we do not explicitly consider the architecture of the computer that will be processing the program or the specific representation that it will assign to real numbers, for example In writing the code, however, most languages require that we make decisions early in the coding about such questions as precision, data representation, and so forth.

We have found one of the major attractions of AUGMENT in writing special-purpose arithmetic packages to be the ability to use abstract (unbound) data types throughout the majority of the programming, binding the data type to a specific representation only in the instructions to AUGMENT and in a few primitive modules of the package

Thus, for example, one might write a package using the data type ETHEREAL, later instructing AUGMENT to convert ETHEREAL to MULTIPLE PRECISION or what have you. Other data types may then be defined as vectors or matrices of ETHEREAL numbers, and AUGMENT will be able to allocate the proper amount of space when it knows the binding of ETHEREAL. Moreover, the routines which manipulate the arrays of ETHEREAL numbers may all be written in terms of operations on ETHEREAL numbers; again, AUGMENT will put everything right at precompile time.

The following sections illustrate this philosophy with concrete examples.

4. THE USE OF AUGMENT IN THE CONSTRUCTION OF THE INTERVAL PACKAGE

The Interval Arithmetic Package described in [7] was motivated by interest in interval arithmetic on the part of several other universities with different computer systems.

The package needed to be flexible enough to accommodate a wide variety of different computer architectures, so we wanted to leave the representation of interval endpoints arbitrary throughout the bulk of the package. But because of FORTRAN's popularity for scientific computation, it was the language of choice for implementing the package. Needless to say, ANSI standard FORTRAN does not have the flexibility we needed in order to accomplish the goals we had set.

We wanted to make the interval arithmetic package easily accessible from the user's point of view. This naturally led us to design the package to be interfaced with AUGMENT But the requirements for flexibility and transportability led us to conclude that the package itself should be written with the aid of AUGMENT

Before we discuss the role of AUGMENT in the implementation of the package, it would be appropriate to include a very brief description of interval arithmetic. The interested reader can find more details in [5].

Interval arithmetic is a means for bounding the error in computation by calculating with pairs of real numbers, the first member of the pair being a lower bound for the true result, and the second an upper bound The foundations for interval mathematics have been carefully laid by Moore [5] and others, so interval mathematics is on firm theoretical ground. There are closed-form formulae for evaluating operations and functions on the space of intervals, so that computation with intervals is reasonably straightforward

In machine interval arithmetic, one naturally represents an interval as a pair of approximate real numbers. In most cases, the existing hardware/software systems are not adequate, for one important reason. in order to preserve the integrity of the interval, calculations involving the lower bound, or left endpoint of the interval, must be rounded downward; those involving the upper bound (right endpoint) must be rounded upward. No production system that we know of provides these roundings.

The design of the arithmetic primitives for the approximate real arithmetic was relatively straightforward; we used the algorithms given in [6] The special functions posed more of a problem: straightforward evaluation of these functions can lead to unacceptably wide intervals. We decided to evaluate these functions in higher precision, and use information about the inherent error in the higher precision procedures before rounding the re-

sults in the proper direction to obtain the desired real approximation.

In order to preserve the desired degree of flexibility, we introduced the nonstandard data type EXTENDED to designate the higher-precision functions, and the nonstandard data type BPA (mnemonic for Best Possible Answer) to designate the approximation to real numbers used for the interval endpoints. The nonstandard data type INTERVAL was then declared to be a BPA array of length 2.

The BPA portion of the package was written in terms of BPA and EXTENDED data types wherever possible. In only a few cases was it necessary to bind BPA to a standard data type in the package modules: such functions as the replacement operator obviously need to be bound to a standard data type to avoid recursive calls.

We illustrate the implementation of the BPA portion of the package with a segment of the BPA square root routine. For simplicity, we have omitted declarations and COMMON blocks which are used to communicate accuracy constants, rounding options, and other information between package modules ACC is an integer variable which indicates the number of accurate digits in the EXTENDED routines. The statement R = ER implicitly invokes the conversion from EXTENDED to BPA, which includes addition or subtraction of an error bound computed from ACC and rounding in the specified direction

```
BPA A, R
EXTENDED EA, ER
EA = A
ER = SQRT(EA)
ACC = IACC(17)
R = ER
```

Next, the INTERVAL portion of the package was written in terms of INTERVAL, BPA, and EXTENDED data types. Here, only three modules are system-dependent.

The following simplified segment of the interval square root routine illustrates the general philosophy used in the implementation of this portion of the package. Declarations and code required for communication with the error-handling routine have been omitted for brevity. Note that before invoking the BPA square root routine (implicitly, twice, once for the right endpoint, or SUP, of the interval, and once for the left endpoint, or INF, of the interval), the variable OPTION is set to specify the desired directed rounding (RDU for upward directed rounding, and RDL for downward directed rounding).

```
INTERVAL A, R
OPTION = RDU
SUP(R) = SQRT(SUP(A))
OPTION = RDL
INF(R) = SQRT(INF(A))
```

Appropriate description decks were prepared for AUGMENT, binding the nonstandard types EXTENDED and BPA to representations in terms of standard data types The entire package was then processed using AUGMENT to extend these bindings.

In order to adapt the resulting package to a different host environment, or different precision, or both, one writes the necessary primitive routines, adjusts the declarations in the description deck as necessary, and reprocesses the package with AUGMENT That this procedure is effective is attested to by the relative ease with which this package was adapted for use on the IBM 370, Honeywell 600, DEC-10, PDP-11, and CDC Cyber systems.

5. ADAPTATIONS OF THE INTERVAL PACKAGE

We discuss two adaptations of the INTERVAL package the first of these is the creation of a package to perform triplex arithmetic, and the second is a package to perform interval arithmetic in multiple precision.

A. THE TRIPLEX PACKAGE: Triplex is a variant of interval arithmetic in which a main, or "most probable", value is carried in addition to the endpoints.

The difference between triplex and interval arithmetic is conceptually quite simple: at the same time one computes an operation or function on the interval endpoints, using the interval mathematics formulas, one evaluates the same operation or function on the main values, using standard real arithmetic, rounding the results to the nearest machine number.

In order to modify the INTERVAL package to perform triplex arithmetic, we needed to add code to all of the interval routines to compute the main values, rename the modules of the package, adjust the formats to accommodate the third value, and, of course, change the representation of intervals to accommodate the main value.

The addition of the extra code was pedestrian; we simply added the appropriate lines of code to each routine to compute the main value. We should note, however, that this did not disturb the existing code, inasmuch as storage and retrieval of the endpoint values had already been defined not in terms of first and second array elements in the interval number, but rather in terms of the field functions INF and SUP respectively (AUGMENT allows the use of such field functions, even when the host FORTRAN compiler does not).

The modules were renamed by suitable use of a text-editing program on the INTERVAL file.

The representation problem was handled simply by changing the word INTERVAL in the type declaration statements to TRIPLEX. No other changes were necessary in the majority of the routines, since AUGMENT automatically extended the new binding throughout the package.

The portion of the triplex square root routine below illustrates the types of changes to the interval package that were necessary to produce the triplex package:

```
TRIPLEX A, R
OPTION = RDU
SUP(R) = SQRT(SUP(A))
OPTION = RDN
MAIN(R) = SQRT(MAIN(A))
OPTION = RDL
INF(R) = SQRT(INF(A))
```

The modification of the INTERVAL package to produce a TRIPLEX package was accomplished in little more than one week of elapsed time, documentation ([1]) excepted.

B. THE MULTIPLE PRECISION INTERVAL PACKAGE: One of the goals of the original design of the INTERVAL package was to facilitate increasing the precision in cases where that was desired. When the multiple precision arithmetic package of Brent [2] became available, it was only natural to consider using that package as a basis for the multiple precision version of INTERVAL

The first step in this process was to develop an AUGMENT interface for Brent's package. This we did in collaboration with Brent.

We are now at the point of developing the multiple precision version of the interval package itself. The steps will be:

(1) Determine the representation to be used for the real approximations. (Brent's package allows a great deal of flexibility in this regard.)

(2) Write the primitive arithmetic operations, basing these on Brent's routines, but providing directed roundings.

(3) Use Brent's package as the EXTENDED arithmetic package

(4) Write the BPA primitives.

(5) Write an additional module which will set the necessary constants based on the run-time precision chosen for the BPA numbers.

(6) Rewrite the description decks as necessary.

(7) Reprocess the package with AUGMENT.

6. OTHER APPLICATIONS OF AUGMENT

In the foregoing, we have illustrated the flexibility that may be gained by using abstract data types. We now consider some extensions of this concept, and some other applications of AUGMENT.

(1) Recursive data type definitions: AUGMENT allows data types to be defined in terms of one another, and this opens up some unique possibilities. The first author once used AUGMENT to aid in the writing of a program to sort multiply-layered information that was stored in the form of trees. This problem was addressed by creating two data types: TREE and NODE One field of a TREE was the root NODE, and one field of a NODE was a TREE. The development of the program using these new data types was straightforward.

(2) Analytic differentiation of FORTRAN functions: This package ([4]) allows one to obtain the Taylor Series expansion or the gradient of a function which can be expressed as a FORTRAN program.

(3) Dynamic precision calculations: In certain types of applications, the precision required for the calculations is a function of the data. AUGMENT allows the definition of data types which are lists, and by using this feature, precision can be determined dynamically.

(4) Simulations: AUGMENT has been used to simulate one computer on another. The technique for doing this is straightforward; one defines a nonstandard data type which represents the simulated machine, and prepares a nonstandard package which copies the arithmetic characteristics and data formats of the target computer.

(5) Algorithm analysis: AUGMENT can be used to provide information such as operation counts in the running of programs or portions thereof One simply defines a nonstandard data type which, in addition to performing the standard operation, increments a counter.

(6) Image processing: The picture processing package developed by W. Fullerton of Los Alamos Scientific Laboratory is one of the most unusual applications of AUGMENT we have yet seen. Various new operators allow the construction of composite pictures from smaller parts, and mathematical functions have even been defined on type PICTURE.

The above illustrations should serve to indicate that the role of AUGMENT in development of mathematical software is limited primarily by the user's imagination.

7. CONCLUSION

We have indicated a number of ways in which the AUGMENT precompiler for FORTRAN can be and has been used to aid in the development of mathematical software. Other applications will undoubtedly be found for this precompiler, since it is both versatile and powerful.

REFERENCES

1. K. Boehmer and J. M. Yohe, A triplex arithmetic package for FORTRAN, The University of Wisconsin - Madison, Mathematics Research Center, Technical Summary Report (to appear).

2. Richard P. Brent, A FORTRAN multiple-precision arithmetic package, Assoc. Comput. Mach. Trans. Math. Software 4 (1978), 57-70.

3. F. D. Crary, A versatile precompiler for nonstandard arithmetics, Assoc. Comput. Mach. Trans. Math. Software (to appear).

4 G Kedem, Automatic differentiation of computer programs, Assoc. Comput. Mach. Trans. Math. Software (to appear).

5. Ramon E. Moore, Interval Analysis, Prentice - Hall, Inc., Englewood Cliffs, NJ, 1966.

6. J. M. Yohe, Roundings in floating-point arithmetic, IEEE Trans. Computers C-22 (1973), 577-586.

7. J. M. Yohe, Software for interval arithmetic. a reasonably portable package, Assoc. Comput. Mach. Trans. Math. Software (to appear).

Extending Reliability: Transformational Tailoring of Abstract Mathematical Software*

James M. Boyle
Applied Mathematics Division
Argonne National Laboratory

EXTENDED ABSTRACT

INTRODUCTION

Over the past decade, mathematical software libraries have matured from small, usually locally-assembled, collections of subroutines to large, commercially-provided libraries which are approaching the status of standards [Aird, Du Croz; Fox]**. Despite the high quality of such libraries and the obvious economic advantages of using routines whose development cost has been shared with many other users, applications programmers, when asked· "Why don't you use routine XYZ from IMSL, or from NAG, or from PORT?" frequently reply that library routines are too general, that they need a routine which takes advantage of special features of their problem, and that since they could not use a library routine without modifying it, they might as well write their own routine from scratch.

In many, if not most, instances, the latter assertion could be easily refuted by a simple competition on selected test problems. However, the need for a routine adapted, or tailored, to a particular problem is more difficult to dismiss. It usually arises from considerations of efficiency, which may range from the perceived inefficiency of the presence of unused options in a routine to the practical impossibility of using a routine whose data representation is utterly incompatible with that needed in the rest of the applications program.

How, then, can mathematical software developers answer this need for mathematical algorithms tailored to individual applications? One approach especially applicable to complicated problems, such as solution of PDE's, is to preprocess a specification of the problem into code which uses a particular software package, as is done in ELLPACK [Rice]. (In some sense, this approach tailors the problem to the software.) For library routines in simpler problem areas, however, it seems necessary to tailor the routine to the problem, since such routines constitute only a small part of the application program, and several routines with possibly conflicting requirements may need to be included. In order for this to be practical, *tailored versions of such routines must be constructed mechanically from very general library routines*. Such mechanical program generation is necessary both to insure that the reliability of the library routine is preserved in its tailored versions and to insure that their construction is not prohibitively expensive [6].

For some time, the TAMPR system has been in use to construct multiple versions, or realizations, of prototype programs for inclusion in mathematical software packages themselves [4,5]. For the LINPACK package, a single prototype routine was used to construct the eight versions representing the combinations of complex or real arithmetic, single or double precision, and calls to Basic Linear Algebra subroutines or in-line code replacements for them [5]

Recent research with TAMPR has focussed on determining the properties a prototype program should have in order to maximize the number and diversity of realizations which can be constructed from it.

ABSTRACT PROGRAMS

The most important property of a prototype program is its *abstractness*. Intuitively, an abstract program captures the essence of a numerical algorithm without cluttering it with irrelevant detail. The presence of irrelevant detail in a prototype program hampers the construction of diverse realizations precisely because a great deal of analysis must be done to verify that it is indeed irrelevant.

The research discussed here has not progressed far enough to characterize abstract programs in general, but examples from the area of linear algebra have been studied sufficiently to illustrate the ideas involved. Consider the code fragment (1):

```
for i = n,1,-1
   for j = i+1,n
      y(i) = y(i)-U(i,j)*y(j)
   end
   y(i) = y(i)/U(i,i)
end
```

and the code fragment (2):

```
for i = n,1,-1
    y(i) = y(i)/U(i,i)
    for j = 1,i-1
       y(j) = y(j)-U(j,i)*y(i)
    end
end
```

Now, both of these fragments actually perform the same computation, the solution of an upper-triangular system of linear equations Uy=x (the final step in the solution of a general linear system whose matrix has been factored into triangular matrices L and U). Fragment (1) is the usual method of solving such a system; it refers to the matrix U by rows. Fragment (2) refers to the matrix U by columns, and is therefore more efficient than fragment (1) on machines with virtual memory or with buffer memory, when the language in which the program is written stores matrices by columns (as does Fortran), see Moler [8] and Smith [10].

Considerable effort is required to see that these two fragments actually perform the same computation; even more effort would be required to devise a way to transform (1) into (2) automatically in order to be able to make the (possibly) more efficient version available to a user. Thus

*Work performed under the auspices of the U.S. Department of Energy.

**Numbers in brackets designate References at end of paper; names in brackets designate authors of other abstracts in this Proceedings.

(1) is not a suitable prototype for tailoring, since it contains difficult-to-discard information about the row-oriented version of the program, which has nothing to do with the specification of the algorithm for the solution of the linear system (see also Smith [10].)

At what level of abstraction, then, is such irrelevant information about how storage is referred to absent from the specification of the algorithm? It is absent when the algorithm is specified in terms of matrix (rather than matrix element) operations. Thus the abstract representation of this algorithm is (3):

$$x = U^{-1}y$$

(Note that this representation of the algorithm is abstract not only with respect to row or column orientation, but with respect to all aspects of the representation of the data; e g., the elements of U could be given by a function.)

TRANSFORMATIONAL SYNTHESIS OF CONCRETE PROGRAMS

This abstract statement of the triangular solution algorithm can be converted into a concrete, executable program by first augmenting it with a specification of the properties of the concrete representations of U,x, and y. The augmented abstract program can then be transformed according to various program-algebraic rules which incorporate the properties into the abstract program and then simplify it where possible (see Boyle [4] and Green [9]).

This process can be illustrated by a sketch of the synthesis of fragment (2) from the abstract program (3). This sketch omits numerous small steps and to avoid introducing unfamiliar notation is presented in terms of quasi-programs. In the actual TAMPR implementation, the transformations are carried out on a representation of the program in an applicative (i.e., expression) language until the final stage, at which Fortran code is generated. As discussed by Backus in his 1977 Turing Lecture [1] such applicative languages have a richer and simpler associated "algebra" than do conventional languages in part because the scope of values is indicated clearly by functional application. (Experience with TAMPR strongly supports this simplicity of applicative languages, which is also well known to LISP programmers.)

The synthesis of fragment (2) depends on some identities from matrix algebra, including·

$$(4)\quad I = \sum_{i=1}^{n} e_i e_i^T$$

where e_i is the i-th unit vector,

$$(5)\quad D + \sum_{i=1}^{n} U' e_i e_i^T = \prod_{i=n}^{1} (I + D e_i e_i^T - I e_i e_i^T + U' e_i e_i^T)$$

where D is an nxn diagonal matrix and U' is upper triangular with zero diagonal; and

$$(6)\quad \prod_{i=n}^{1} (I + D e_i e_i^T - I e_i e_i^T + U' e_i e_i^T)$$

$$= \prod_{i=1}^{n} (I + D e_i e_i^{-1} e_i^T - I e_i e_i^T - U' e_i e_i^T D^{-1} e_i e_i^T)$$

The idea of a matrix A being "stored by columns" is thus expressed as

$$A = AI = A\sum_{i=1}^{n}(e_i e_i^T) = \sum_{i=1}^{n} ((Ae_i)e_i^T),$$

Ae_i is the i-th column of A.

The synthesis begins with (3) augmented to indicate that U is an upper-triangular nxn matrix stored by columns, that y is an n-vector, and that x is to be identified with y:

```
    y = U**-1*y
--> y = (diag(U) + uppersubtri(U))**-1*y
```

now U is expanded by columns:

$$\text{--> } y = (\text{diag}(U) + \sum_{i=1}^{n} (\text{uppersubtri}(U)e_i e_i^T))^{-1} y$$

$$\text{--> } y = (\prod_{i=n}^{1}(I+(\text{diag}(U)e_i)e_i^T - Ie_i e_i^T + (\text{uppersubtri}(U)e_i)e_i^T))^{-1} y$$

$$\text{--> } y = \prod_{i=1}^{n}(I+(\text{invdiag}(U)e_i)e_i^T - Ie_i e_i^T - (\text{uppersubtri}(U)e_i)(e_i^T\,\text{invdiag}(U)e_i)e_i^T) y$$

```
--> for i = n,1,-1
      y = y + (invdiag(U)e_i)(e_i^T y) - e_i(e_i^T y)
      -(uppersubtri(U)e_i)(e_i^T invdiag(U)e_i)(e_i^T y)
    end
```

(Note that the above program is the point of departure for a "vector" solution of the triangular system, although this problem is not particularly well suited to vector computation) Now expand the remaining vectors to components (the assignment is still a vector one):

```
--> for i=n,1,-1
      Σ(j=1..n) e_j(e_j^T y) = Σ(j=1..n) e_j(e_j^T y) + e_i(e_i^T invdiag(U)e_i)*
      (e_i^T y) - e_i(e_i^T y) - Σ(k=1..n) (e_k(e_k^T uppersubtri(U)e_i)*
      (e_i^T invdiag(U)e_i)(e_i^T y))
    end
```

After a number of steps which include determining that a vector temporary is not required in converting the vector assignment to a scalar one, the component-level is reached·

```
--> for i = n,1,-1
      for j = 1,i-1
        (e_j^T y) = (e_j^T y) - (e_j^T uppersubtri(U)e_i)*
        (e_i^T invdiag(U)e_i)(e_i^T y)
      end
      (e_i^T y) = (e_i^T invdiag(U)e_i)*(e_i^T y)-
      (e_i^T uppersubtri(U)e_i)*(e_i^T invdiag(U)e_i)(e_i^T y)
      for j = i+1,n
```

$(e_j^T y) = (e_j^T y) - (e_j^T \text{ uppersubtri}(U) e_i)*$

$(e_i^T \text{ invdiag}(U) e_i)(e_i^T y)$

end

end

Uppersubtri(U) implies e_k^T uppersubtri(U)$e_j = 0$ for $k \geq j$, assignments of the form x = x need not be carried out, and common subexpressions can be computed once, so that the program becomes:

```
--> for i = n,1,-1
        t = (e_i^T invdiag(U)e_i)(e_i^T y)
        for j = 1,i-1
            (e_j^T y) = (e_j^T y) - (e_j^T uppersubtri(U)e_i)*t
        end
        (e_i^T y) = t
    end
```

The temporary can be eliminated by reordering and the component references converted to conventional form to obtain fragment (2), above. Thus the transformational synthesis of a program takes place in a large sequence of small steps, each effected by a transformation based on a relatively simple mathematical theorem or axiom.

CORRECTNESS OF CONCRETE PROGRAMS

As discussed in [4], transformationally-constructed concrete programs inherit the correctness of the abstract prototype program provided the transformations themselves are correct. A correct transformation may add information to the abstract program, but this information must be consistent with the properties assumed for the abstract program. (In this sense, the process is rather like constructing the integers as a "concrete" instance of a ring, by augmenting the ring axioms with additional axioms consistent with the original set.) Thus anything provable about the abstract program remains true for any of the concrete realizations of it.

The proof that an arbitrary set of transformations is correct may be difficult in general. However, as discussed in [7], if each transformation in the set is itself "semantics-preserving" (i.e., replaces a part of a program with program text which does not contradict the meaning of the original text), the correctness of the transformational process is guaranteed (if it terminates). Usually it is quite easy to see that an individual transformation is semantics-preserving, especially when it is based on a mathematical property.

Finally, the fact that the abstract program is closer to the mathematical formulation of a problem than is an ordinary program means that its correctness is much easier to prove. In the present example (but not in general) the abstract program and its specification are almost identical; about the only thing which must be verified is that the product and assignment involve consistently-dimensioned arrays.

Incidentally, the fact that the concrete realizations of (3) do not cause out-of-bounds subscript references when executed follows from the fact that (3) involves consistently dimensioned arrays and the fact that those transformations which introduce subscripts also simultaneously introduce the index sets for them based on the array dimensions. (See Backus [1], section 5, for some discussion of the significance of this.) This two-stage proof is much easier than showing directly that (2) does not execute out-of-bounds subscript references. The difference is even more dramatic for an abstract Gaussian elimination algorithm and a realization of it employing implicit pivoting; the subscripts in the latter program are themselves subscripted variables, and it is very difficult to prove directly from the code that they are always in bounds.

WHY TRANSFORMATIONS?

It is perhaps interesting to conclude by posing the questions: Why use a program transformation system to construct concrete realizations of abstract programs? Why not simply devise an extended language for specifying abstract programs and a processor to translate it into an existing language (e.g., EFL [Feldman] and Bayer and Witzgall's Complete Matrix Calculi [3]) or directly into machine language? Or, why not implement by hand a relatively fixed ensemble of routines for different data representations and call them as appropriate to a particular user's needs?

Clearly, these alternatives are not completely distinct, for the "processor" for an extended language might consist of a collection of transformations, while some transformations insert code which could be thought of as very small subroutines. However, what I call a program transformation system is distinguished from the other two approaches primarily because it provides a high-level notation for specifying and applying source-to-source program transformations and because it can manipulate any programming-language construct (not just subroutines and arguments). Transformation systems of this type include not only TAMPR, but also those proposed by Bauer [2], and by Loveman and Standish (see [6]).

In my experience, the idea of providing for abstract program specification through a fixed extended language is too static an approach to be effective. The work discussed here is far from complete, yet already it has undergone numerous revisions. Had a particular notation been fixed, or had the transformations been implemented in a fixed processor, they would have been very difficult to modify. Moreover, emphasis on Designing a Language tends to cause one to get lost in a tangle of language-related issues which are not very germane to abstract program specification; indeed the expression and function notation available in slightly modified Fortran or in Algol seems quite adequate for experimentation. Finally, even extensible languages, which permit the definition of new data types and operators (e.g., Algol 68), do not usually provide a means for easily specifying optimizations (especially global ones) for these extensions. As we have seen, such optimizations are both fundamental and rather specific (e.g., the row analog of (6), which shows that n instead of n(n+1)/2 divisions suffice) and it is unreasonable to expect them to be built into a general-purpose language processor. Specifying these optimizations by transformations not only allows them to be easily tested and modified, it also permits them to be selectively applied to classes of programs which may reasonably be expected to use them.

Similarly, the implementation, by hand, of a set of subroutines tailored to various properties is also static and not very reliable; moreover, the set needed is very large, being the product of the number of variables, the number of representations of each, etc. In a transformational formulation, the number of transformations needed behaves more like the sum (plus some initial overhead). Thus, use of transformations enables one to manage the complexity of the problem and thereby greatly enhances reliability.

CONCLUSION

I have sketched how various concrete executable programs can be constructed automatically from an abstract prototype program by applying transformations based on theorems of matrix algebra and on "algebraic" properties of programming languages. Although this research has just begun, it offers the hope of being able to provide a user with highly efficient programs tailored to his environment while maintaining the advantages of high reliability and low cost associated with routines from the best mathematical software libraries. Moreover, the transformations which produce such programs themselves represent a valuable resource: a formal codification of rules for writing linear algebra programs.

ACKNOWLEDGMENTS

Work on the derivation of row and column oriented programs from an abstract prototype was begun by Brian Smith in conjunction with Janet Bentley while Brian was on sabbatical at the NAG Central Office, Oxford. This work was supported by NAG; preliminary results are reported in [10]. I am indebted to Brian for numerous discussions which helped the work discussed here to evolve into its present form.

REFERENCES

1. J. Backus, Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs, Comm. ACM 21, 8, Aug. 1978, 613-641.
2. F. L. Bauer, Programming as an Evolutionary Process, Proc. 2nd Int'l Conf. on Software Engineering, San Francisco, 1976, 223-234.
3. R Bayer and C. Witzgall, Some Complete Calculi for Matrices, Comm. ACM 13, 4, April 1970, 223-237.
4. J M. Boyle, Mathematical Software Transportability Systems -- Have the Variations a Theme? in Portability of Numerical Software, Lecture Notes in Computer Science, No. 57, Springer-Verlag, 1977.
5. J. M. Boyle and K. W. Dritz, three papers on the TAMPR system in J. R. Bunch, Ed., Cooperative Development of Mathematical Software (available from the authors).
6. J. M. Boyle, K. W. Dritz, O. B. Arushanian, and Y. V. Kuchevskiy, Program Generation and Transformation -- Tools for Mathematical Software Developement, Information Processing 77, North Holland 1977, 303-308.
7. J. M. Boyle and M. Matz, Automating Multiple Program Realizations, Proc. of the MRI Symposium, XXIV: Computer Software Engineering, Polytechnic Press, 1977, 421-456.
8. C. B. Moler, Matrix Computations with FORTRAN and Paging, Comm. ACM 15, 4, April 1972, 268-270.
9. C. C, Green, The Design of the PSI Program Synthesis System, Proc. 2nd Int'l Conf. on Software Engineering, San Francisco, 1976, 4-18.
10. B. T. Smith, Portability and Adaptability -- What are the Issues? in D Jacobs, Ed., Numerical Software -- Needs and Availability, Academic Press, 1978, 21-38.

D9

N79-12724

FORTRAN TOOLS

by

Leon Presser
Softool Corporation
340 South Kellogg Avenue
Goleta, Calif. 93017

ABSTRACT

This paper outlines an integrated set of Fortran tools that are commercially available. The basic purpose of various tools is summarized and their economic impact highlighted. The areas addressed by these tools include: code auditing, error detection, program portability, program instrumentation, documentation, clerical aids and quality assurance.

THE PURPOSE OF THIS PRESENTATION is to outline a number of powerful software tools presently marketed by Softool Corporation. Here we shall only discuss an integrated set of Fortran tools, since the Fortran language and the portability of Fortran programs is a key to the development of numerical software. The issues, however, extend well beyond numerical software, and apply to software development in general.

A perspective is in order. The current cost of software development is exorbitant and the quality of the products generated leaves much to be desired. The single most serious issue facing us today is the lack of coherent methodologies for the construction of software. Such methodologies should address the entire construction process from requirement analysis to maintenance. Furthermore, the absence of explicit methodologies explains the lack of sound software management disciplines, so necessary in a process of such complexity as the construction of software.

The key to methodology and management are proper tools. Indeed, Webster's New Collegiate Dictionary defines management as: 'judicious use of means to accomplish an end', and it defines tool as 'a means to an end'. From these we deduce: (software) management: 'judicious use of (software) tools.'

FORTRAN TOOL SET

The available Fortran tools apply principally to the programming, testing and maintenance phases of the software construction process. These tools consist of:

- standard auditors
- error detectors
- portability aids
- instrumenters
- documenters
- clerical aids
- quality assurers

All of the Fortran tools are coded in a highly portable subset of Fortran. Next, we highlight some of the existing Fortran tools.

1. ANSI FORTRAN CHECKER & ERROR DETECTOR

This tool accepts as input a Fortran source program and outputs clear documentation pinpointing:

- deviations from ANSI (1966) standard
- errors present in the source program
- portability problems
- management indices

To substantiate the economic advantages that result from use of this tool let us simply focus on its error detection capability. Our experience indicates that during software development this tool reduces the number of compilations necessary to obtain a program ready for execution by a factor of 3 to 5. Use of this tool for the analysis of production programs (i.e., programs that have been running 'correctly' for years) generates about 3 definite errors per thousand lines analyzed! [1,2]. If we assume, quite conservatively, that a person costs $20/hour and that each problem present in the software will require 8 person-hours for its removal, we have:

$$\frac{3 \text{ problems}}{1000 \text{ lines}} \times \frac{8 \text{ hours}}{1 \text{ prob.}} \times \frac{\$20}{1 \text{ hr.}} \simeq \$500/1000 \text{ lines}$$

That is, use of this tool would save on the order of $500 for each 1000 lines of Fortran processed!

Concerning portability, this tool detects and documents a large number of potential Fortran portability problems, such as:

- statement orderings,
- operations of equal precedence that are not fully parenthesised,
- implicit conversions,
- side effects, and many others

We have been moving software across different computers in a straightforward manner with the aid of this tool.

Each Softool tool produces appropriate management indices that help concerned management obtain visibility over their software. For example, this tool generates three different indices:

- average number of errors per statement
- average number of warnings per statement
- average number of portability problems per statement

2. FORTRAN INSTRUMENTER I

This tool accepts as input a Fortran source program and execution time test data sets for the program. Upon execution of the source program it automatically generates routine level profiles. These profiles quantize testing and optimization efforts in detail. The information provided by the profiles includes: percent of

test coverage, time spent in each routine, number of calls to each routine, test effectiveness index and an optimization index.

This tool has a major impact in expediting and reducing the effort spent in testing. In essence, it helps minimize the test data required for a specified coverage, which in turn results in decreased test data execution time and also reduces the human time required to analyze test results. Similarly, this tool is a great aid in focusing optimization efforts. Our experience indicates that savings in excess of 10% of the overall software development effort are readily obtained with the help of this tool.

This tool also serves as a valuable adjunct to the ANSI FORTRAN CHECKER AND ERROR DETECTOR during program portability efforts.

3. FORTRAN DOCUMENTER A

The main purpose of this tool is to facilitate and expedite the uniform documentation of Fortran source program units. In essence, management provides to this tool the definition of a documentation template. Programmers write their code in a simple and straightforward shorthand format. The code written by the programmers is input to this tool which outputs fully commented units of source code, documented according to the predefined documentation template. Our experience indicated that excellent, self-contained documentation can be consistently obtained with the aid of this tool. Moreover, the keypunching and/or terminal entry work is reduced by a factor of about 5!

Other members of our integrated set of Fortran tools are a statement level instrumenter and two other documenters that accept as input a source program and generate extensive local and global cross-reference directories.

SUMMARY

The objective of this presentation has been to outline a set of integrated Fortran tools available from Softool Corporation. These tools have extensive and highly cost-effective application in the development, management and quality assurance of Fortran based software. If we are to conquer the ubiquitous software problem we must promptly incorporate into our methodology tools of the kind described here.

REFERENCES

1. A Comparative Analysis of the Diagnostic Power of Commercial Fortran Compilers. Softool Corporation, Report No. F001-10-77, October 1977.

2. A Comparative Analysis of Programming Methodologies with the Aid of the Ansi Fortran Checker and Error Detector. Softool Corporation, Report No. F002-2-78. February 1978.

USING TOOLS FOR VERIFICATION, DOCUMENTATION AND TESTING[1]

Leon J. Osterweil
Department of Computer Science
University of Colorado
Boulder, Colorado 80309

I. Introduction

There has been considerable interest lately in methodologies for the production of high quality computer software. Work in this area has been carried out by researchers in a wide variety of disciplines and covers an impressive spectrum of approaches. Some of the more active current lines of research include software management techniques [1, 2], creation of error resistant programming techniques [3, 4, 5]; and design of error resistant programming languages [6, 7]

There has also been considerable activity in the creation of program testing, verification and documentation tools. The work in this area has been directed primarily towards two different but related goals -- the detection and examination of errors present in a program, and the determination that a given program has no errors of some particular type. Among the diverse activities in this area, this paper shall focus on four of the major approaches -- namely dynamic testing, symbolic execution, formal verification and static analysis In this paper, the different patterns of strengths, weaknesses and applications of these approaches will be shown. It will, moreover, be demonstrated that these patterns are in many ways complementary, offering the hope that they can be coordinated and unified into a single comprehensive program testing and verification system capable of performing a diverse and useful variety of error detection, verification and documentation functions.

II. Four Error Detection and Verification Techniques

In dynamic testing systems, [8, 9, 10, 11] a comprehensive record of a single execution of the program is built. This record -- the execution history -- is usually obtained by instrumenting the source program with code whose purpose is to capture information about the progress of the execution. Most such systems implant monitoring code after each statement of the program. This code captures such information as the number of the statement just executed, the names of those variables whose values had been altered by executing the statement, the new values of these variables, and the outcome of any tests performed by the statement. The execution history is saved in a file so that after the execution terminates it can be perused by the tester. This perusal is usually facilitated by the production of summary tables and statistics such as statement execution frequency histograms, and variable evolution trees.

Many dynamic testing systems also monitor each statement execution checking for such error conditions as division by zero and out-of-bounds array references The monitors implanted are usually programmed to automatically issue error messages immediately upon detecting such conditions in order to avoid having the errors concealed by the bulk of a large execution history.

Some systems [9, 10] even allow the tester to create his own monitors, direct their implantation anywhere within the program, and specify where and how their messages are to be displayed. The greatest power of these systems is derived from the possibility of using them to determine whether a program execution is proceeding as intended. The intent of the program is captured by sets of assertions about the desired and/or correct relation between values of program variables.

Dynamic testing systems provide strong error recognition and exploration capabilities, but are unable to determine the absence of errors. Their results are narrowly applicable, being valid only for a single program execution. These results are quite extensive and detailed, however, providing sufficient material for deep insight into the program. These systems allow extensive human interaction, and their power is most fully realized when a skilled human tester is using them interactively. They require as input a complete set of actual program input data. The success of a dynamic testing run as a vehicle for discovering and exploring errors is largely dependent upon the selection of revealing and provocative input data. This usually presumes the involvement of a human tester who is knowledgable about the program being tested.

In symbolic execution, symbolic representation (in the form of formulas) are kept for the evolving values of variables instead of numeric quantities. For a given path through the program, the values of all the variables encountered are maintained as formulas. The only unknowns in these formulas are the input values to the program; all other values of variables are functions of constants and these input values and, therefore, can

[1] Research supported by NSF Grant # MCS77-02194.

be removed by substitution. The formulas can be examined by a human tester to see whether they embody the intent of the program If so, then the tester has determined that the program will yield the desired results for all executions which follow the given program path. A number of symbolic execution systems have been produced [12, 13, 14, 15].

Clarke's system [12] is perhaps the most interesting symbolic execution system in the context of this paper, in that it indicates better than the others the range of error detection and verification capabilities possible with the symbolic execution approach. In Clarke's system, the execution path which is specified as input is used to dictate the required outcome of all conditional tests along the path. Hence, the path dictates a set of constraints which must be satisfied in order for execution to proceed along the given path. These constraints are in terms of current values of program variables, but through the use of symbolic execution, they can more profitably be expressed as relations in terms of current values of program variables. The system of relations obtained in this way is taken to be a set of simultaneous constraints, and is examined by Clarke's system for consistency. A solution to a consistent set of constraints is a set of values which, when taken as input to the program, will force execution of the given path If the constraints are inconsistant, then the path is unexecutable -- that is, there exists no data which will effect the execution of the given path.

Clarke's system also creates additional, temporary constraints for the purpose of error detection and verification. Constraints are created which test for the possibility of array bounds violations, DO statement loop control variable errors and division by zero. Clarke's system will attempt to solve the system of constraints to produce program input data which forces the traversal of the given input path, followed by a zero-divide error at the given point.

Symbolic execution systems provide strong error detection capabilities and some pathwise verification capabilities which fall short of the power of full verification Symbolic execution systems provide diagnostic information which is applicable to classes of executions rather than a single execution. This is achieved by supplying symbolic relationships between program values in place of precise numeric data. These systems require human intervention and evaluation in order to carry out error detection, although the pathwise validation capabilities require no human assistance. Symbolic execution systems require that a test path through the program be supplied. It is important that the path given be revealing and provocative, thur requiring the skills of a knowledgable human tester

In static analysis systems, the text of a source program is examined in an attempt to determine whether the program is defective due to local malformations, improper combinations of program events, or improper sequences of program events. In order to make this determination, each statement of the program is represented by a small, carefully selected set of characteristics. The static analysis system can then examine each characteristic set on a statement-by-statement basis for malformations, and various combinations and sequences of statements on a characteristic-by-characteristic basis for faulty program structure or coordination. No attempt is made at replicating the entire behavior or functioning of the program Rather, static analysis attempts to examine the behavior of the entire program only with respect to certain selected features

The syntax checking of individual statements of a program provides a good example of static analysis. More interesting and valuable error detection is obtained by examining the characteristics of combinations of statements. For example, illegal combinations of types can be detected by examining declaration statements and then examining the executable statements which refer to the variables named in the declarations Similarly, mismatches between argument lists and parameter lists associated with the invocation of procedures or subroutines can also be made by static analysis systems. Some of the types of static analysis discussed above are available with most compilers. Other types, such as argument/parameter list agreement are far less common in compilers, but are found in such stand-alone static analysis systems as FACES [16] and RXVP [17].

The use of static analysis techniques to examine sequences of program events enables the detection of still other types of program errors. In DAVE [18] each statement of a program is represented by two lists -- a list of all variables used to supply values as inputs to the computation, and a list of all variables used to carry away values produced as output by the computation. The static analysis then examines sequences of statement executions which are possible given a program's control flow structure, and determines such things as whether it is possible to reference an uninitialized or otherwise undefined variable, and whether it is possible to compute a value for a variable and then never refer to the computed value In such cases, the static analyzer determines and outputs the statement sequence for which the anomalous pattern of references and definitions occurs. Similarly, it would be possible to scan programs for other improper sequences of events such as openings, writings, and closings of files; and enablings and disablings of interrupts. Paths along which these sequences could occur would then also be determined. It should be emphasized here that the most recent static analysis systems which examine event sequences for improprieties employ search techniques which enable the examination of all sequences of statement executions which are possible, given the flow of control structure of the program. These search techniques, first studied in connection with program optimization [19, 20, 21, 22] are also quite efficient. Unfortunately, the most efficient of them will merely detect the existence of such improper sequences. Somewhat less efficient algorithms are needed in order to determine the actual sequences.

It can be seen from the preceding paragraphs that static analysis systems offer a limited amount of error detection, but are capable of performing certain verification functions. Static analysis only examines a few narrow aspects of a program's execution, but the results of this analysis are comprehensive and broadly applicable to all possible executions of the program Here, as in the case of symbolic execution, it is seen that the verification capabilities are obtained without the need for human interaction. A human tester is required, however, in order to interpret the results of the analysis and pinpoint errors. Finally, it is important to observe that static analysis requires no input from a human tester. As output, it produces either paths along which anomalous program behavior is possible, or validation results indicating that no anomaly-bearing paths exist

In formal verification, the code comprising a program is compared to the total intent of the program, as captured and expressed in the form of assertions. Assertions are used to describe the program output expected in response to specified program inputs. The goal of the formal verification is to prove a theorem stating that the program code actually achieves this asserted input/output transformation. The proof of this theorem is reduced to the proof of a coordinated set of lemmas. The statements of these lemmas are derived from a set of intermediate assertions positioned in specific locations throughout the program code These assertions describe precisely the desired status of program computations at the locations of the assertions. Differences in status between assertion sets separated in position by a body of code embody the transformation which that code segment is intended to perform. Proving that the code segment achieves the transformation establishes the lemma that the segment is correct. A total formal verification is achieved if the program is also proven to always terminate.

It is quite significant to observe that symbolic execution is the technique used to determine the transformation effected by a given code segment. Hence, the symbolic execution technique is central to formal verification Formal verification can, in fact, be viewed as a formalized framework for carrying out a rigorously complete and coordinated set of symbolic executions and comparisons to intended behavior.

Formal verification is the most rigorous, thorough and powerful of the four techniques presented here. There are sizable problems in carrying it out, however. The size and intricacy of the work make it costly. The need for exact mathematical models of the desired and actual behavior of a program invite errors and weakening inaccurate assumptions. It is generally agreed, however, that the discipline and deep perception needed to undertake formal verification are useful in themselves Anticipation of formal verification seems to foster good program organization and design Attempts at formal verification invariably lead to improved insight into both the goals and implementation of a program.

III. An Integrated Testing, Analysis and Verification System

Recently, each of the four above techniques has received considerable attention and investigation. Stand-alone systems, implementing each have been constructed, and experience has been gained in using each. Partly as a result of this experience, there is a growing concensus that no single technique adequately meets all program testing verification and analysis needs, but that each contributes some valuable capabilities It thus becomes clear then that the four techniques should not be viewed as competing approaches, but rather that each offers useful but different capabilities. Attention then naturally turns to the examination of how the various capabilities can be merged into a useful total methodology and system.

Such a methodology is described now. The methodology makes provision for the progressive detection and exploration of errors as well as provision for selective verification of different aspects of program behavior.

Both types of activities are begun with static analysis of the source program using a pathwise anomaly detecting analyzer. In the next phase of the methodology, symbolic execution is used to further the results of static analysis. The symbolic execution focuses on anomaly-bearing paths detected by the static analyzer to further the error detection and verification power of the methodology. The methodology next calls for the application of either dynamic testing or formal verification. Dynamic testing is used to obtain the most precise but restricted examination of the nature and sources of errors and anomalies whose existence has been determined during the first two phases. Symbolic execution is used to generate test data for the dynamic test of individual cases Formal verification is used to obtain the most definitive demonstration of the absence of errors Extreme rigor and thoroughness can be applied at high cost in showing the absence of errors.

A schematic diagram of the methodology is shown in Figure 1

The above strategy organizes the four techniques into a progression of capabilities which is natural in a number of important ways. It begins with a broad scanning procedure and progress to deeper and deeper probing of errors and anomaly phenomena It initially requires no human interaction or input. It progresses to involve more significant human interaction as human insight becomes more useful in tracing errors to their sources and constructing mathematical demonstrations of correctness. It provides the possibility of some primitive verification without human intervention, and then allows error detection based upon the negative results of the verification scan. The flow of data is also most fortuitous. The first phase static analysis requires no input. It produces as output, however, paths through the program which are deemed to be significant in error

output of the first phase. Finally, the dynamic testing phase requires actual program input data It has been observed, however, that symbolic execution systems can be used to produce data sets which are sufficient to force the execution of their input paths Hence, the second phase can be used to provide the input required by the dynamic testing phase.

It is also interesting to observe that the use of assertions provides a unifying influence in integrating the four techniques. All techniques except static analysis are explicit assertions to demonstrate either the presence or absence of errors Static analysis uses implicit assumptions of proper behavior as embodied in language semantics, but could also benefit from explicit assertions Seen in this light, the four techniques basically differ in the manner and extent to which they perform assertion verification. Thus, it seems reasonable to require that a program and initial set of assertions be submitted. The adherence of program to assertions would be examined at every stage. The stages would test the adherence in different ways, progressively establishing firmer assurances of adherence or focusing more sharply on deviations.

IV. Conclusion

The foregoing section has presented a rather saguine view of the capabilities of an integrated testing system combining the best features of static analysis, symbolic execution, dynamic testing, and formal verification Although software systems implementing each of these techniques have been produced, the task of constructing a usable system is still far more formidable than simply building software interfaces between existing systems Significant research must be completed before a useful system can be built.

The outlines of some of the longer range outcomes of this research can be observed already. It appears, for example, that this research will show that many of the testing operations currently performed by dynamic testing systems alone, can be performed more effectively by some combination with static analysis, symbolic execution and formal verification. This would lessen the reliance of testing upon chance and human interaction. It also appears that this research will show that the activities of program testing and formal verification are more closely related than previously generally thought. Some of the static analysis techniques proposed here can reasonably be thought of as techniques for producing proofs of the correctness of certain restricted aspects of a given program Moreover, certain proposed applications of symbolic execution are tantamount to assertion verification over a limited range. It is expected that this research may provide some insight into some ways in which testing and proving activities can be utilized as complementary activities The proposed research should confirm these and other important conjectures.

V. Acknowledgments

The author would like to thank Lloyd D. Fosdick for the many valuable and stimulating conversations which helped shape the ideas presented here, as well as for his perceptive comments on early versions of this paper. The ideas presented here were also shaped by stimulating conversations with Lori Clarke, Bill Howden, Jim King, Don Reifer, Dick Fairley, Leon Stucki, Bob Hoffman, and many others

VI. References

[1] F T Baker, "Chief Programmer Team Management of Production Programming," IBM Systems Journal (11) pp 56-73 (1972).

[2] D S. Alberts, "The Economics of Software Quality Assurance," AFIPS Conference Proceedings (45) pp. 433-442 (1976 National Computer Conference).

[3] E W. Dijkstra, "Notes on Structured Programming", in Structured Programming by O.J Dahl, E.W. Dijkstra and C.A.R. Hoare, Academic Press, London and New York, 1972

[4] N. Wirth, "Program Development by Stepwise Refinement" CACM 14, pp. 221-227 (April 1971).

[5] D.L. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules" CACM 15, pp. 1053-1058 (December 1972).

[6] N. Wirth, "An Assessment of the Programming Language PASCAL" IEEE Transactions on Software Engineering SE-1, pp 192-198 (June 1975).

[7] J D. Gannon and J.J. Horning, "Language Design for Program Reliability", IEEE Transactions on Software Engineering SE-1, pp. 179-191 (June 1975)

[8] R.A Balzer, "EXDAMS· Extendable Debugging and Monitoring System", AFIPS 1969 SJCC 34 AFIPS Press, Montvale, New Jersey, pp. 567-580

[9] R.E. Fairley, "An Experimental Program Testing Facility", Proceedings of the First National Conference on Software Engineering, IEEE Cat. #75CH0992-8C, pp 47-52

[10] L.G. Stucki and G.L. Foshee, "New Assertion Concepts for Self Metric Software Validation", Proceedings 1975 International Conference on Reliable Software, IEEE Cat. #75CH0940-7CSR, pp. 59-71.

[11] R. Grishman, "The Debugging System AIDS", AFIPS 1970 SJCC 36 AFIPS Press, Montvale, N.J., pp. 59-64.

[12] L. Clarke, "A System to Generate Test Data and Symbolically Execute Programs", IEEE Transactions on Software Engineering SE-2, pp. 215-222 (September 1976)

[13] W.E. Howden, "Experiments with a Symbolic Evaluation System", AFIPS 1976 NCC 45, AFIPS Press, Montvale, N.J , pp. 899-908.

[14] J.C. King, "Symbolic Execution and Program Testing", CACM 19, pp. 385-394 (July 1976).

[15] R.S. Boyer, B Elspas, and K.N. Levitt, "SELECT--A Formal System for Testing and Debugging Programs by Symbolic Execution", Proceedings 1975 International Conference on Reliable Software, IEEE Cat #75CH0940-7CSR, pp. 234-245

[16] C V Ramamoorthy and S.B F. Ho, "Testing Large Software with Automated Software Evaluation Systems", IEEE Transactions on Software Engineering SE-1, pp. 46-58.

[17] E.F. Miller, Jr., "RXVP, Fortran Automated Verification System", Program Validation Project, General Research Corporation, Santa Barbara, California (October 1974).

[18] L.J. Osterweil and L.D. Fosdick, "DAVE--A Validation, Error Detection, and Documantation System for Fortran Programs", Software Practice and Experience 6, pp. 473-486

[19] E.F. Allen and J. Cocke, "A Program Data Flow Analysis Procedure", CACM 19, pp. 137-147 (March 1976).

[20] K.W. Kennedy, "Node Listings Applied to Data Flow Analysis", Proceedings of 2nd ACM Symposium on Principles of Programming Languages, Palo Alto, California, pp. 10-21 (January 1975).

[21] M S. Hecht and J.D. Ullman, "A Simple Algorithm for Global Data Flow Analysis Problems," SIAM J. Computing 4, pp. 519-532 (December 1975).

[22] J.D. Ullman, "Fast Algorithms for the Elimination of Common Subexpressions", Acta Informatica 2, pp. 1910213 (December 1973).

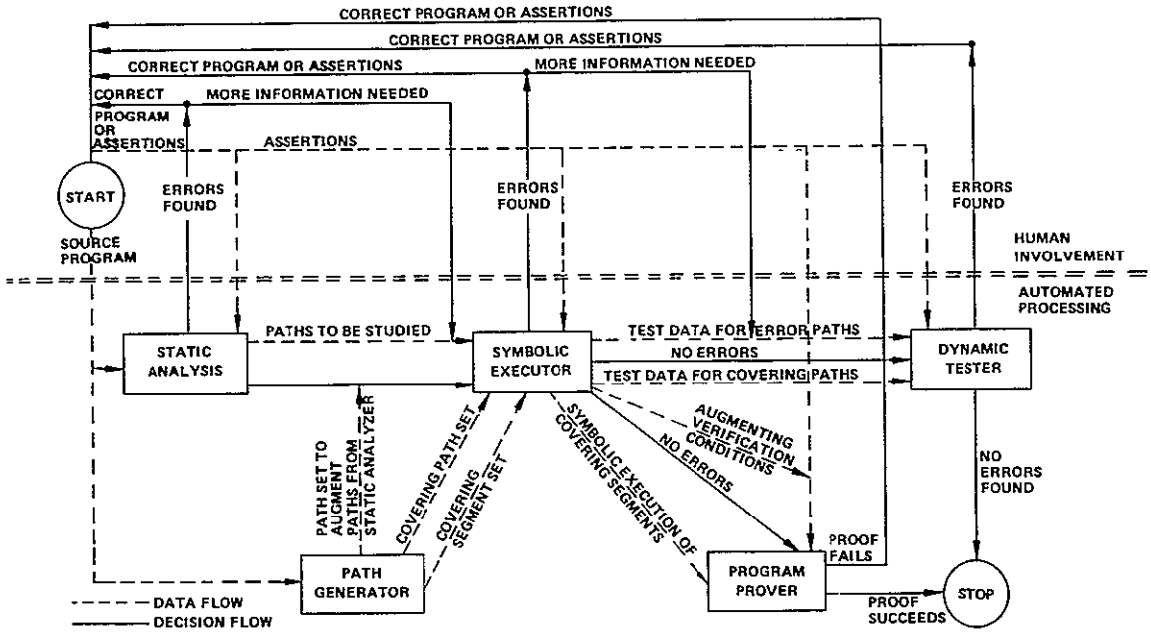


Figure 1. A Diagram of the Proposed System

A CASE FOR TOOLPACK

Webb Miller, Department of Mathematics

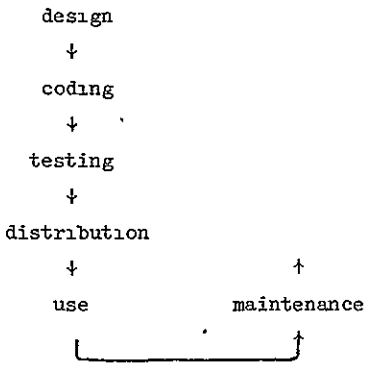
University of California
Santa Barbara, California, U.S.A.

ABSTRACT

We propose a collaborative effort to develop a systematized collection of software tools for Fortran programming.

1. SOME SOFTWARE TOOLS

LET US TRY TO CLASSIFY some software tools in terms of their applicability during the development of a typical Fortran program.



DESIGN - At this phase the program's specifications are determined and algorithms and data structures are selected. Software tools can assist in the preparation and checking of design specifications. (12)* A much more specialized tool is software for roundoff analysis (14,15) which can be used to help design a program performing matrix computations.

CODING - Text editors, compilers, structured Fortran preprocessors like RATFOR (11) or EFL, and source-text formatters like POLISH (3) are useful.

TESTING - Debugging systems (8) help track down program errors and profilers (10,22) can locate untested code and code in need of scrutiny for possible manual optimization. The PFORT Verifier (20) and DAVE (6,17,18) can diagnose certain kinds of errors. Currently, much research (9,16,19,22) in software engineering is focused on this phase of software development.

DISTRIBUTION - Distribution aids can assist in preparing multiple versions (for different machine number systems or different machines) from the "master version" of a program. (2, pp. 305-423)

MAINTENANCE - Distribution aids can also be used to keep track of the various releases of a program and to update collections of programs stored on tape. (5,21) For maintaining a program written by someone else, it may help to automatically convert Fortran into, say, well-structured Fortran 77 with a structurer. (1)

2 TOOLPACK

We feel that the time has arrived for a collaborative effort to develop a systematized collection of software tools for Fortran programming. Since we envision an effort along the lines of the "PACK series" (EISPACK and its descendants) we use the name "TOOLPACK "

Probably the first and most crucial design decision would be the selection of a Fortran extension, which we will call FE, with adequate control structures and with data types sufficient to make writing the software tools relatively painless (the tools should be written in this language). Compatability with Fortran 77 seems extremely desirable, but perhaps other considerations will make, say, EFL the proper choice A processor for FE should be designed with all of its possible uses in tools (i)-(iv) in mind (we could also use it to update our minicompiler (14,15))

The exact contents of TOOLPACK would have to be dictated by the willingness and interests of contributors One possibility is that TOOLPACK initially contain:

(i) a FE-to-PFORT translator,
(ii) a FE source-text formatter which transforms PFORT into PFORT,
(iii) a FE profiler,
(iv) distribution and maintenance aids.

Of the candidates for eventual inclusion in TOOLPACK, some are promising, e.g., DAVE (we hope that improvements will make wide use economically feasible) and program structurers (they are of interest as much for insights into the programming process gained during their development as for their usefulness as a tool). On the other hand, some candidates are of doubtful value, e.g , automatic test data generators (we feel that our method (13) is best in many cases, but even it does not seem especially useful) and symbolic execution systems (9), while some candidates seem to have no immediate practical value, e.g., automatic program verifiers. (4)

3. JUSTIFICATION

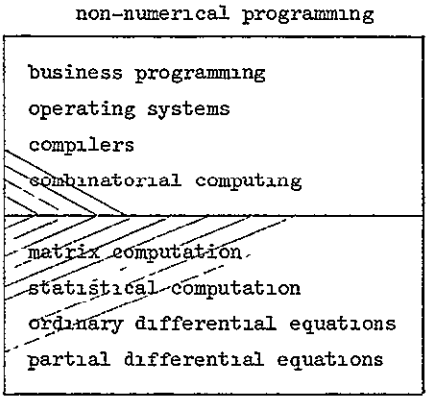
We are convinced that TOOLPACK would easily pay for itself in savings to Fortran programmers. However, there are software development issues at stake which are far more important

The PACK series is the prime example of the spread of what might be called the "PACK paradigm"

*Numbers in parentheses designate References at end of paper.

of computer science research Characteristically, (i) such endeavors take the form of collaboration by research leaders to produce portable state-of-the-art software and all levels of supporting documentation, (ii) federal funding provides partial support and (iii) the resulting software and documents are placed in the public domain.

Consider the following map of the computer programming milieu (it is neither complete nor drawn to scale). The approximate domain of the PACK paradigm is shaded



Actually, the existence of the pictured beachhead for the PACK paradigm in non-numerical territory is questionable. The examples we have in mind, in particular the non-numerical contributions in the Collected Algorithms of the ACM, are not collaborative on a scale even approaching that of the PACK series.

There are numerous reasons why the PACK paradigm will never be the universal model of software development. Some of the difficulties are unavoidable. For instance, combinatorial computing resists "black boxes" since typically it is hard to detach general-purpose combinatorial programs from the ad hoc data structures which are used.

Some of the impediments to the spread of the PACK paradigm are less intrinsic. A major culprit is the reward system in both academic and business sectors. Duplication of effort often results in greater total reward than does collaboration. Moreover, the spread of the paradigm to non-numerical territory is impeded by the scarcity (or complete absence) of precedent-setting examples.

The PACK paradigm should be fostered in those areas of software development where it is appropriate. TOOLPACK would do much to encourage its spread by producing programs which can be readily appreciated by non-numerical programmers

4. PROPOSALS

If sufficient support materializes for TOOLPACK, then by mid-November a proposal should be submitted to the Department of Energy and the National Science Foundation to jointly fund a 2- or 3-day TOOLPACK workshop to be held in, say, March. The proposal might request expenses (up to $500 per person) for 10-12 workshop participants and $1000 overhead to cover secretarial assistance (before and during the workshop), Xeroxing, telephoning and postage.

Attendance at the workshop should be limited to persons likely to make a commitment to TOOLPACK, either as principal investigators developing software or as test site coordinators. The goal of the workshop should be to come as near as possible to completing a package of proposals for constructing and testing TOOLPACK, also to be submitted to the Department of Energy and the National Science Foundation jointly. Some discussion of design questions (e.g , the choice of FE) should take place, but only until someone assumes responsibility for a generally approved software tool so that a concrete proposal can be made. Detailed design questions need not be considered until TOOLPACK development is funded.

ACKNOWLEDGMENT

We would like to thank Stan Brown, Don Johnson and especially Fred Krogh for their helpful comments

REFERENCES

1. B Baker, "An Algorithm for Structuring Flowgraphs." JACM 24, 1977, pp. 98-120

2. W. Cowell (ed.), "Portability of Numerical Software." Springer Lecture Notes in Computer Science #57, 1977.

3. J Dorrenbacher et al., "POLISH, a Fortran Program to Edit Fortran Programs." Dept. of Computer Science Report CU-CS-050-74, University of Colorado, 1974.

4. B. Elspas et al., "An Assessment of Techniques for Proving Program Correctness " Computing Surveys 4, 1972, pp. 97-147.

5. S. Feldman, "Make--a Program for Maintaining Computer Programs." Computer Science Technical Report 57, Bell Labs , 1977.

6. L. Fosdick and L. Osterweil, "Data Flow Analysis in Software Reliability." Computing Surveys 8, 1976, pp. 305-330

7. A. Hall, "SEDIT--a Source Program Editor " Computing Science Technical Report 16, Bell Labs., 1974.

8. A. Hall, "FDS a FORTRAN Debugging System, Overview and Installer's Guide." Computing Science Technical Report 29, Bell Labs., 1975.

9 W. Howden, "An Evaluation of the Effectiveness of Symbolic Testing." Software--Practice and Experience 8, 1978, pp. 381-397

10. D. Ingalls, "The Execution Time Profile as a Programming Tool." In R. Rustin (ed.), "Design and Optimization of Compilers." Prentice-Hall, 1972, pp. 107-128.

11 B Kernighan, "RATFOR--a Preprocessor for a Rational Fortran." Software--Practice and Experience 5, 1975, pp. 395-406

12. H. Kleine, "Software Design and Documentation Language." JPL Publication 77-24, Jet Propulsion Laboratory, 1977.

13. W. Miller and D. Spooner, "Automatic Generation of Floating-Point Test Data " IEEE Trans. on Software Engineering SE-2, 1976, pp. 223-226.

14. W. Miller and D. Spooner, "Software for Roundoff Analysis II." TOMS, 1978.

15. W. Miller and C. Wrathall, "Software for Roundoff Analysis of Matrix Computations." 1979.

16. L. Osterweil, "A Proposal for an Integrated Testing System for Computer Programs." Dept. of Computer Science Report CU-CS-093-76, University of Colorado, 1976

17. H. Osterweil and L. Fosdick, "DAVE--a Validation, Error Detection and Documentation System for Fortran Programs." Software--Practice and Experience 6, 1976, pp. 473-486.

18. L. Osterweil and L Fosdick, "Some Experience with DAVE--a Fortran Program Analyzer." Proc. National Computer Conference, 1976, pp. 909-915

19. C. Ramamoorthy and S.B. Ho, "Testing Large Software with Automated Software Evaluation Systems." IEEE Trans. on Software Engineering 1, 1975, pp. 46-58.

20. B. Ryder, "The PFORT Verifier." Software--Practice and Experience 4, 1974, pp. 359-377

21. W.V. Snyder, "A Transportable System for Management and Exchange of Programs and Other Text." Talk at this conference.

22. L. Stucki and A. Foshee, "New Assertion Concepts for Self-Metric Software Validation." In IEEE Proc. International Conf. on Reliable Software, 1975, pp. 59-65.

D12

N79-12727

Programmable Formatting of Program Text: Experiences Drawn from the TAMPR System*

Kenneth W. Dritz, Applied Mathematics Division
Argonne National Laboratory
Argonne, Illinois, U.S.A.

EXTENDED ABSTRACT

The TAMPR System originated as an approach to the problem of automating the routine modifications of Fortran source programs required to adapt them to a variety of uses or environments [1]** Overall, the system accomplishes such modifications by applying transformations to Fortran programs at the source level. But the process differs markedly, in detail, from string-based editing or macro expansion. Three steps are involved:

(1) A Fortran source program is processed by the TAMPR Recognizer, yielding essentially a parse tree called the abstract form.
(2) The Transformation Interpreter applies IGT's (Intragrammatical Transformations) to the abstract form as tree operations [2]
(3) The abstract form is then reconverted to source program form by the Formatter.

By ensuring that the transformations are applied only to the correct syntactic entities and only in the intended contexts, the use of the abstract form greatly simplifies establishing the reliability of the overall process.

The Formatter, of course, is responsible for meeting the requirements of the target language, such as use of blanks, statement continuations, etc. In addition, the formatting process is charged with imparting to the resultant program a certain degree of style. Areas of concern here are spacing between symbols, choosing "logical" breakpoints in multi-line statements, and, at a higher level, commenting and indentation to help reveal program structure The expectation of variation of opinion among researchers as to what constitutes good style plus the knowledge that our own criteria for good style would evolve with experience led us to reject the idea of a fixed formatting process, such as that essentially used in Lang's STYLE editor [3]. The Formatter, accordingly, was designed to interpret an easily modified set of formatting instructions.

Several alternative designs were possible for the Formatter. For example, the approach of embedding formatting directives in the program to be formatted, as in document formatting systems, was rejected partly because it would have seriously complicated the application of IGT's. More importantly, however, the idea of formatting instructions separate from the programs appealed because it would permit one set of instructions to be used for the formatting of many Fortran programs. Pursuing that idea, we next concentrated on the form of the instructions. A tabular encoding in the manner of Koch and Schwarzenberger [4] suffered in our opinion from being obscure, unnatural, and not sufficiently expressive and was rejected. We chose instead to develop a high-level procedural language, Format Control Language, containing appropriate application-oriented features as well as general computational capability. The Formatter, then, is programmed in FCL.

How does one write FCL programs that are broadly applicable to the conversion of a whole class of Fortran programs from abstract to concrete form? We will answer this question by surveying some of the features of FCL.

Since the trees to be formatted are essentially parse trees, a first-order approximation to the desired output can be obtained simply by traversing the tree in canonical (left-to-right, or recursive descent) order producing output at the terminal nodes. That behavior is in fact built in to the Formatter and does not require programming in FCL. The automatic traverse falls short of providing all the required behavior, however For instance, no special treatment would be provided for label definitions or for "end-of-statement," and spacing between tokens would be fixed at some value, say zero or one. Or there could be various reasons (see [5]) for pruning the tree (i.e., for not descending into certain subtrees) or for emitting some computable function of the "print name" of one or more terminal nodes rather than just the print names themselves, in order and with the fixed spacing between them. These problems are addressed by various features of FCL, some of which are described below.

In order to intercept the automatic traverse at any desired point one employs a fundamental FCL control structure called a *production block*. A production block is similar to a procedure block except that its "name" is a representation of a production in the grammar to which the tree conforms (in other words it is the representation of a type of node which may occur in the abstract form tree). Continuing the analogy, a production block is invoked not by a CALL statement but by arrival of the automatic traverse at a node of the type described by the production block's name

Within the body of a production block, the individual components of its name can be used to refer symbolically to the actual node at which the traverse was interrupted and to its immediate subnodes. Two useful and essential functions derive from this capability. First, the "print names" of terminal subnodes may be used in *EMIT statements* to produce fragments of textual output, or they may be used in computations leading to such output, for example, to construct and emit the *n*H*text* form of a Hollerith constant represented in the tree by a terminal node whose print name contains just the *text* part. Second, the automatic traverse may be continued into a particular subtree headed by a subnode of the node at which it was interrupted by referring to the subnode in a *FORMAT statement*.

Normal block structure rules, when applied to the nesting of production blocks, yield another useful and essential feature. Since a nested

*Work performed under the auspices of the U.S. Department of Energy.
**Numbers in brackets designate References at end of paper.

block is not "known" and cannot be invoked unless its containing block is active, we are provided with a simple and natural way to make the formatting of a particular type of node (e.g., phrase) dependent on its context. An example of the use of this feature, in conjunction with *SPACE statements*, is to replace the default spacing of one as it applies, for instance, on either side of an arithmetic operator by zero when that operator appears inside a subscript list.

The problem of breaking and continuing statements that are too long to fit on one line has received special attention Statements can in general be thought of as composed of *nested lists*. For example, a logical-IF statement containing, say, a CALL statement is a list of two items, the IF-test and the CALL statement. The argument list in the call is a list of expressions separated by commas. Each expression is a list of terms separated by "+" or "-", each term is a list of factors separated by "*" or "/", etc. When it is determined that a statement must be broken, it is broken at the rightmost breakpoint of the shallowest (highest level) unclosed list that has at least one breakpoint on the line, and it is continued on the next line in the column in which the broken list began. Since the beginnings, ends, and breakpoints of lists do not always bear constant relationships to the recursive phrase structure of the grammar, we require that they be marked in passing by the execution of FCL statements provided for that purpose. For instance, the beginning of an argument list is just inside the left parenthesis, while its breakpoints are just after the commas and its end is just outside the right parenthesis (by not closing the list until after the right parenthesis, a possible "dangling parenthesis" is avoided if the line should overflow by just that one character) Some controversy surrounds the following question: if a list of terms or of factors, etc., is broken, should the breakpoint be *before* or *after* the arithmetic operator? The programmability of the Formatter gives the TAMPR user his choice

Early in the design of the Formatter a more general approach to choosing the breakpoints was discussed, namely, that of buffering an entire statement and then choosing all of its breakpoints to minimize the overall badness of the result, defined in some suitable way At the time we were not prepared to deal with that much added complexity. Perhaps we will restudy that approach after Knuth shares the results of implementing a similar dynamic programming algorithm for the line division of paragraphs in his technical text system, TEX [6].

FCL contains also a standard assortment of general control structures and declarative and computational features for integer and character string data types. These have proved of particular value in algorithms to detect and preserve columnar relationships and paragraph structure in blocks of comments that are subjected to a variety of substitutions and other minor transformations. A discussion of other features of FCL, such as those for marking labels and ends of statements, setting off comments, and indenting, may be found in [5], along with numerous illustrations of the use of all the features mentioned in this abstract.

The Formatter's programmability, particularly its general computational and control capabilities, has aided its own evolution by permitting new ideas to be simulated, at least, for evaluation before taking the decision to build them in as primitives. It has likewise made possible the use of TAMPR in new application areas (see [7]), such as a verification condition generator in a program verification project. In that use, the requirement to format ordinary infix logical and arithmetic expressions as prefix output was easily met.

REFERENCES

1. J. M. Boyle and K. W. Dritz, An Automated Programming System to Facilitate the Development of Quality Mathematical Software, Information Processing 74, 542-546, North-Holland Publishing Company, 1974.

2 J. M. Boyle and M. Matz, Automating Multiple Program Realizations, Proceedings of the M.R.I. International Symposium XXIV: Computer Software Engineering, 421-456, Polytechnic Press, Brooklyn, N.Y., 1977.

3 D. E Lang, STYLE Editor: User's Guide, Department of Computer Science Report CU-CS-007-72, University of Colorado, Boulder, Colorado, 1972.

4. K. Koch and F. Schwarzenberger, A System for Syntax-Controlled Editing of Formula Text, Proceedings of the Newcastle Seminar on Automated Publishing, 1969.

5. K. W. Dritz, An Introduction to the TAMPR System Formatter, in J. R. Bunch (ed.), Cooperative Development of Mathematical Software, 89-103, Department of Mathematics Technical Report, University of California, San Diego, California, 1977.

6. D. E. Knuth, Mathematical Typography (Gibbs Lecture, AMS), Report STAN-CS-78-648, Computer Science Department, Stanford Univ , February 1978.

7. K. W. Dritz, Multiple Program Realizations Using the TAMPR System, in W. R. Cowell (ed.), Proceedings of the Workshop on the Portability of Numerical Software, 405-423, Springer-Verlag, Berlin, 1977.

New Languages for Numerical Software

Eric Grosse

Stanford University
Stanford, California

1. Introduction

Existing languages for numerical software are not altogether satisfactory. Fortran, even preprocessed, has troublesome limitations Unfortunately, proposed replacements have been discouragingly large, or omit essential features like variably dimensioned arrays and Fortran compatibility

A new language has been designed to include such crucial features, but otherwise be as small as possible This language, called T, includes indention to indicate block structure, novel loop syntax, and engineering format for real constants By preprocessing into PL/I, implementation cost was kept low.

But this language can only be regarded as a stopgap. The next step is to deal with more fundamental issues, by more fully exploiting arrays, by making pointers available in a controlled way, and by arranging for better declaration and syntax for using subroutine libraries.

2. Defects of Existing Languages

Why do we need new languages? Consider how your favorite programming language deals with the following catalog of annoying features encountered in writing mathematical software Though the list is long, a failing on even one issue can make life rather unpleasant.

Arithmetic

- Converting a program from single to double precision is a chore Variables must be declared, constants like 1E0 converted to 1D0, and function names changed
- Precision can easily be lost accidentally Intermediate results may be truncated, or extended with garbage.
- One must remember a distinct name for each version of each routine that handle different types and precisions of numbers.
- Silent type conversions, as in I = J = 0, can give wrong answers without warning.

Arrays and Pointers

- Subscript checking is not available.
- Array bounds are fixed at compile time. Working with several datasets requires repeated recompilation or the use of a clumsy stack mechanism
- All lower bounds must be 1 If the natural origin is 0, off-by-one errors easily slip in.
- Information about bounds must be explicitly passed in subroutine calls. This is so inconvenient that fake bounds of 1 are used instead.

- References are used in a fundamental but unnatural way, and therefore wind up being hidden. (e g. **real** really means **ref real**)
- Operations like the sum of two vectors or the sum of all components of a vector are not available, or only available by a somewhat unreadable subroutine call. Or the operations are available, but their implementation is expensive enough that one is advised not to use them.

Input-Output and Error-Handling

- The underflow routine, as it properly sets the result to 0, loudly reports its activity.
- When a fatal error occurs, the input-output buffers are not flushed No subroutine call traceback is provided either
- The free-format facilities are nonexistent or produce ugly output, so awkward format specifications must be prepared

Restrictions

- The form of identifiers is severely limited
- Do loop bounds are restricted.
- Modern control constructs, macros, procedure variables, and fully-supported structured data types are missing
- The interface to Fortran routines is weak, particularly with arrays, input-output, and error handling
- There are not enough restrictions to allow good optimization, particularly when function calls are present

Clutter

- Quotation marks are required around keywords Semicolons must appear in specified places.
- Poor comment conventions make it tricky to comment out blocks of code.
- The reference manual is bulky, hard to read, and obsolete.
- The compiler is slow, expensive, large, difficult to use, and contains many bugs

3. A Quick Fix

Each of these problems has been solved before, but not in a single language. So to show it could be done and to provide a useful tool, I drew up plans for a small language T [Grosse 1978] and, with the help of another numerical analysis student, implemented it in about a month.

We chose to preprocess into PL/I, following Kernighan and Plauger's [1976] motto to "let someone else do the hard work " By incorporating a macro processor to handle tasks like reversing array subscripts for Fortran compatibility, we managed to avoid parsing most program text, in the same spirit as RATFOR and other Fortran preprocessors At the same time we cleaned up a few aspects of PL/I, for example converting 1 0 automatically to BINARY (1 00000000...) so that precision and type troubles are avoided

In addition, we were able to include several unusual features For the ALGOL veteran, perhaps the most striking is the complete absence of BEGINs and ENDs. Not only is the text indented, but the indention actually specified the block structure of the program. Such a scheme was apparently first proposed by Landin [1966], except for an endorsement by Knuth [1974], the idea seems to have been largely ignored

Ideally, the text editor would recognize tree-structured programs In practice, text editors

tend to be line oriented so that moving lines about in an indented program requires cumbersome manipulation of leading blanks. Therefore the current implementation of T uses BEGIN and END lines, translating to indention on output.

Whatever the implementation, the key idea is to force the block structure and the indention to be automatically the same, and to reduce clutter from redundant keywords.

In addition to normal statement sequencing and procedure calls, three control structures are provided. The CASE and WHILE statements are illustrated in this typical program segment

```
WHILE(NORMYP > 1(-3) & 1<=IFLAG & IFLAG<=3 )
    TOUT  = T + 10(-3)/NORMYP
    ODE(DF,2,Y,T,TOUT,RELERR,ABSERR,IFLAG,ODEWORK,ODEIWORK)
    CASE
        2 = IFLAG
          GDRAW (Y,PF)
        3 = IFLAG
          PUT('ODE DECIDED ERROR TOLERANCES WERE TOO SMALL.')
          PUT ('NEW VALUES')
          PUT DATA (RELERR,ABSERR)
        ELSE
          PUT ('ODE RETURNED THE ERROR FLAG ')
          PUT DATA (IFLAG)
    FIRST
    DF(T,Y,YP)
    NORMYP  = NORM2(YP)
```

The CASE statement is modelled after the conditional expression of LISP, the boolean expressions are evaluated in sequence until one evaluates to YES, or until ELSE is encountered The use of indention makes it easy to visually find the relevant boolean expression and the end of the statement

One unusual feature of the WHILE loops is the optional FIRST marker, which specifies where the loop is to be entered In the example above, the norm of the gradient, NORMYP, is computed before the loop test is evaluated Thus the loop condition, which often provides a valuable hint about the loop invariant, appears prominently at the top of the loop, and yet the common n-and-a-half-times-'round loop can still be easily expressed.

The FOR statement adheres as closely as practical to common mathematical practice

```
FOR ( 1 <= I <= 3 )
    NORMSQ  = (Y(1)-X(I,1))**2 + (Y(2)-X(I,2))**2
    Z  = Z + H(I)*EXP(-0 5*W(I) *NORMSQ)
```

Several years experience with these control constructs has demonstrated them to be adequately efficient and much easier to maintain than the alternatives.

Beginners often find Fortran's input/output the most difficult part of the language, and even seasoned programmers are tempted to just print unlabelled numbers, often to more digits than justified by the problem, because formatting is so tedious. PL/I's list and data directed I/O is so much easier to use that it was wholeheartedly adopted in T By providing procedures for modifying the number of decimal places and the number of separating blanks to be output, no edit-directed I/O is needed Special statements are provided for array I/O so that, unlike PL/I, arrays can be printed in orderly fashion without explicit formatting

Since almost as much time is spent in scientific computation staring at pages of numbers as at pages of program text, much though was given to the best format for displaying numbers.

In accordance with the "engineering format" used on Hewlett-Packard calculators and with standard metric practice exponents are forced to be multiples of 3 This convention has a histogramming effect that concentrates the information in the leading digit, as opposed to splitting it

between the leading digit and the exponent, which are often separated by 14 columns. the use of parentheses to surround the exponent, like the legality of imbedded blanks, was suggested by mathematical tables. This notation separates the exponent from the mantissa more distinctly than the usual E format

4. A Longer-Term Solution

By building on a rather powerful host language, T goes a long way towards meeting the standards implied in section 2 But there are certain fundamental problems that will probably stimulate a completely independent implementation.

Source-level optimization is desirable because particular transformations can be performed or analyzed by hand. To permit this and to clarify the use of "arbitrary" arguments for passing information untouched through library routines, a controlled form of pointers can be introduced By manipulating descriptor blocks, considerably more powerful array operations are feasible. The increasing availability of computer phototypesetting equipment has implications for language syntax Declarations and statements ought to be able to be intermixed. With the growing importance of subroutine libraries, provision must be made for language extensions to support new data types and operators.

By using Fortran very carefully and invoking verification tools, it is now possible to write programs that run, without any change whatsoever, on practically any computer This extreme portability can probably never be achieved by any new language Even a portable Fortran preprocessor requires some effort to bring up at a new site But I believe that the advantages of instant portability are overwhelmed by the expressiveness, efficiency, and clean environment that new languages can provide

5. References

E Grosse, *Software Restyling in Graphics and Programming Languages,* Stanford University, STAN-CS-78-663, [1978].

B. W. Kernighan, and P Plauger, *Software Tools,* Addison Wesley, [1976].

D. Knuth, *Structured Programming with Goto Statements,* Computing Surveys 6, 261-301, [1974].

P Landin, *The Next 700 Programming Languages,* Comm ACM 9, 157-166, [1966].

D14

N79-12729

The Programming Language EFL

S. I. Feldman

Bell Laboratories
Murray Hill, New Jersey 07974

Introduction

EFL is a comprehensive language designed to make it easy to write portable, understandable programs It provides a rich set of data types and structures, a convenient operator set, and good control flow forms The lexical form is easy to type and to read

EFL was originated by A D Hall The current author completed the design of the language and wrote the current compiler Whenever possible, EFL uses the same forms that Ratfor [1] does, in this sense EFL may be viewed as a superset of Ratfor EFL is a well-defined language, this distinguishes it from most "Fortran preprocessors" which only add simple flow of control constructs to Fortran

The EFL compiler generates (possibly tailored) Standard Fortran as its output EFL should catch and diagnose all syntax errors

The following description will be brief and informal Many details of the language are omitted The reader is assumed to be familiar with Fortran, but not necessarily with Ratfor

Syntax

EFL is line-oriented The end of line terminates a statement unless the last token on the line is an operator or is part of a test or loop construct A sharp (#) starts a comment, which continues till the end of the line Statements may be terminated by a semicolon, in that case more than one statement may appear on a line EFL uses natural characters (&, <,) rather than Fortran's multi-character identifiers (AND , LT ,) for operators Variable names begin with a letter, and may be followed by any number of digits and letters Numeric and logical constants follow the rules of Fortran Character constants are surrounded by quotes

Program Form

Every procedure (main program, subroutine, function) begins with a **procedure** statement and finishes with an **end** statement

Macros may be given values in a **define** statement

```
define EOF -4
define BUMP { i +=1 ; j +=1 }
```

A file may be read in with a line like

```
include filename
```

Data Types

The basic data types of EFL are **integer, logical, real, complex, long real** (=double precision), and **character** The first five are the usual Fortran data types Character data are strings of fixed length, represented in Hollerith strings EFL also has homogeneous aggregates (arrays) and inhomogeneous aggregates (structures) Arrays may have more than one dimension, and lower bounds need not be 1 In declarations, attributes may be factored, and more than one attribute may be declared on a line Initial values may also be given on the declaration line

```
character(8) greeting = "hello"
integer size = 5*9
```

The following single statement declares a common block

```
common(x)
    {
    logical firsttime
    character(7) name
    array(0  99)
        {
        integer flag
        complex value
        }
    }
```

The block contains a logical variable, a character variable, and two arrays, each containing 100 elements

Structures may be made up of objects of different types A structure may be given a tag, that tag acts as a type name (analogous to **integer**) in the rest of the program Thus,

```
struct point
    {
    integer color
    real x,y
    }
```

declares a shape. Later declarations might be

```
point  p, z(50)
struct
    {
    integer ptr
    point p
    }
        buffer(100)
```

The latter declares a variable with an unnamed shape **buffer** is an array of structures containing points as elements An element of an array is selected by the usual subscript notation; subscripts may be arbitrary integer expressions

```
a(i,j) = b( max(i,j) )
```

Elements of structures are specified by giving the element names

```
xcoord = buffer(5) p.x
```

Structures may be passed to procedures There is also a mechanism for dynamic location of structures.

Operators

The usual arithmetic operators (+, −, *, /, **) logical operators (&, |, ~), and relational operators (<, <=, >, >=, ==, ~=) are provided Quantities of different types may be mixed within expressions In addition to ordinary assignment (=), there are a number of operating assignment operators

```
k +=1
q &= p
```

are equivalent to the Fortran statements

```
k = k + 1
q = q and p
```

Assignments are expressions Multiple assignments are expressed directly

```
a = b = c
```

is equivalent to

```
b = c
a = b
```

Assignment expressions are very useful for capturing values·

```
if( (k = inchar(unit)) == "x")
    . .
```

invokes the function **inchar**, stores its value in **k**, then compares it to the letter *x*.

In addition to the usual logical operators, there are two sequentially evaluated ones, && and || The expression E_1 && E_2 is evaluated by first evaluating E_1, only if it is true is E_2 evaluated Thus, these operators guarantee the order of evaluation of logical expressions

EFL provides generic functions and a general type conversion mechanism EFL chooses the correct version of the intrinsic functions depending on the argument types **sin**(5 1d1) generates a call to the **dsin** function

Flow of Control

EFL statements are normally executed in sequence Statements may be grouped within braces This is the only grouping mechanism. The testing and looping constructs usually operate on a single statement; a braced group acts as a single statement The usual **if** forms are permitted

```
if(a < b)
    a = b

if(a < b)
    {
    x = a
    y = b
    }
else
    {
    x = b
    y = a
    }
```

There is also a **switch** statement for branching on many values

```
switch( inchar(unit) )
    {
    case 0
        eof = 1
        done()
    case 1
        letter = inval
    case 2
        digit = inval
    }
```

For looping, a Fortran **do**, a conventional **while** and **repeat** · **until**, and a general for loop are provided The **do** is unnecessary but very convenient

```
do i = 1,n
    a(i) = b(i) + c(i)

while( (k = inchar(unit)) != EOF)
    {
    a(i) = inval
    i += 1
    }

repeat
    x += delta
    until( ( delta = phi(x)) < epsilon)
```

The for statement is borrowed from the C[2] language It has three clauses an initial value, a test, and a step rule The loop

```
for(p=first, node(p) ptr>0, p=node(p) ptr)
    out( node(p) value )
```

will output every element of a linked list

```
struct
    {
    integer ptr
    real value
    }
        node(100)
```

Statements may be labeled and reached by a **goto** Labels are alphabetic identifiers There are (optionally multilevel) **break** and **next** statements for leaving a loop, or going to the next iteration These statements are needed only rarely, but very convenient occasionally

The **return** statement exits from a procedure It may have a function value as argument

```
return( sin(x+1) )
```

Miscellaneous

The discussion above touches on most of the features of the languages There are also input/output statements in the language that give access to Fortran's I/O, but in a somewhat more convenient form.

A number of statement forms are included to ease the conversion from Fortran/Ratfor to EFL These atavisms include numeric labels, computed **goto** statements, **subroutine** and **function** statements, and the ability to use the AND , etc forms under compiler option.

Compiler

The current version of the compiler is written in C Its output is readable Fortran To the extent possible, variable names are the same as in the EFL text Statement numbers are consecutive, and the code is indented (This is possible because of the two-pass nature of the compiler)

There are EFL compiler options for tailoring the output for a particular machine or Fortran compiler Implementation of character variables requires knowledge of the packing factor (number of characters per integer) The default output formats are machine-dependent, as is the handling of input/output errors Except for issues of these sorts, the output is machine-independent The EFL compiler worries about generating Standard Fortran, following

its restrictions on line format, subscript form, DO statement syntax, type agreement, and so on; the EFL programer need know nothing of these rules

References

1 B W Kernighan and P. J Plauger, *Programming Tools,* Prentice-Hall, 1976

2. B. W Kernighan and D M Ritchie, *The C Programming Language,* Prentice-Hall, 1978

D15

N79-12730

DESIGN PRINCIPLES OF THE PORT LIBRARY

Phyllis Fox

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

The Bell Laboratories Mathematical Subroutine Library, PORT, has been under development at Bell Laboratories for the past few years The design of PORT stemmed from the basic principles of portability and ease of use. The attributes and mechanisms used in the library to support this philosophy include the use of a portable subset of Fortran, and the use of (machine-dependent) functions to supply the necessary environment parameters. Abbreviated calling sequences are made possible by a simplified error-handling technique, and by the use of a portable stack allocator for temporary workspace. Our experience to date with these approaches, and our plans for the future are reviewed here

BELL LABORATORIES MATHEMATICAL SUBROUTINE LIBRARY, PORT, is a library of portable Fortran programs for numerical computation An article describing the library appeared in a recent issue of TOMS [1] together with CACM Algorithm 528 [2] containing the basic utilities for the library - the functions defining the machine constants, the error handling and the storage stack structure Our rallying points throughout the development of the library have been portability, modularity and economy of structure, all of which lead to simplified calling sequences.

PORTABILITY

The rules of the game for programs acceptable to PORT require that (1) the program pass the PFORT Verifier [4], which guarantees that it is written in a portable subset of ANSI Fortran (1966 version), and (2) any machine-dependent quantities used in the program be obtained by invoking the appropriate one of the three PORT functions that return integer, real or double-precision values

The machine-dependent values for the constants in these functions are set when the library is installed on the computer. The same tape is sent to all computer sites; the recipient simply removes the C's in column 1 from the data statements defining the constants for the computer at hand, and compiles the library Values are defined on the PORT tape for the Burroughs 5700/6700/7700 systems, the CDC 6000/7000 series, the Cray 1, the Data General Eclipse S/200, the Harris SLASH 6 and SLASH 7, the Honeywell 600/6000 series, the IBM 360/370 series, the Interdata 8/32 the PDP-10 (KA or KI processor), the PDP-11 and the UNIVAC 1100 series PORT has been installed on each of these computers

ERROR HANDLING

The design of PORT's error-handling has been described elsewhere (e g. [1]). Briefly, two types of errors are specified - fatal and recoverable. Errors which can be checked a priori by the user, such as the length of a vector not being negative, are typed fatal. All others are recoverable, but revert to fatal unless the user specifically elects to enter the "recovery" mode and deal with the errors as they occur. The method has proved to be safe for the inexperienced user, but to allow flexibility and leeway for the expert.

The error handler is heavily used within the library itself As a matter of policy, outer level routines called by the user reinterpret any errors occurring at lower levels. Thus the user never sees an error message from a mysterious lower-level routine Everything is aboveboard

STORAGE STACK

The concept of a dynamic storage stack, as implemented in a labeled COMMON region in the PORT library is also described in [1]. Each program that needs scratch space, allocates space on the stack for the computation, and then returns the space when it is done The stack-handling routines are, of course, available to the user as well, and have been found to provide efficiencies of space usage for all concerned

(Note that the design of the error-handling and of the stack means that neither error flags, nor scratch vector designators appear in the calling sequences to PORT routines.)

A stack dump routine, written by Dan Warner, has been tremendously useful in debugging; it will be included in a future edition of PORT

CURRENT ACTIVITY

PORT is now installed at 15 sites within Bell Laboratories, its use at Murray Hill on the Honeywell computer fluctuates between about 160 and 200 accesses per day It has been sent to 29 external sites including locations in Austria, Belgium, Egypt and the Netherlands, and several requests are currently in process.

One of the more interesting uses to which the library has been put in the last few months has been in computer benchmarking using a set of computationally intensive programs The benchmarking programs are based on the PORT library, so that once the library is installed, they run easily without adaptation It has been reassuring to find that at those sites where the library was not already installed, it could be put up in about an hour.

An on-going task is the development of various categories of test programs for the library We have recently developed a collection of the example programs listed in the documentation in the PORT Users Manual [3], These will be included on the tape when PORT is sent out to requesting sites. On another front, Norm Schryer has developed very probing and exacting tests of the PORT utilities; these are included on the Algorithm 528 tape [2]. Finally, tests for all the routines in the library are being developed or assembled (In most cases they already exist and have been in use for some time)

Our primary activity, however, still centers around the construction of high quality portable numerical software. Linda Kaufman is providing the library with routines in the areas of linear algebra and least-squares fitting In the pde direction PORT will acquire Norm Schryer's PEST package for one-dimensional partial differential equations coupled with ordinary differential equations, and Jim Blue's Laplace equation program based on a boundary integral equation formulation will be incorporated into the library Dan Warner has a good linear programming program, and other things are coming along You might say - the future looks bright on the port side

REVIEW OF DESIGN PRINCIPLES

In developing a program to go into PORT we consider it a matter of pride as well as principle to "pipe" it through a sequence of processes untouched by human hands (if not minds) The original is probably written in the language, EFL, discussed by Stu Feldman elsewhere in these proceedings. The EFL version is automatically translated into Fortran, and then is put through the PFORT Verifier to check for language. Then after a, shall we say variable amount of time required for debugging, the double-precision version of the program is automatically twinned to a single-precision version and both are put on the tape The initial comment section of the program must contain a detailed description which can be transcribed into the file containing the phototypeset original for the program reference sheets for the Users Manual An example program, illustrating the use of the subroutine, is developed, tested, and automatically added to the reference sheet file as well as to the set of example programs on tape Admittedly, at the moment, the transcription from program comments to reference sheets is done somewhat manually, but the macros used to form this phototypesetting version make the process very quick We intend to increase the level of automation

Historically, we have found it best to exercise our programs on a number of prob-

lems within Bell Laboratories before issuing them outside. During this period any difficulties that come up get incorporated into a growing set of tests for the programs.

SUMMARY OF DESIGN PHILOSOPHY

Our current thinking about mathematical program libraries leads us to wonder whether a new kind of duplication of effort won't occur if the several libraries now extant try to become all things to all people We may not wish to make PORT an all-inclusive library covering the multiple facets of numerical and statistical computation, but rather to maximize our result/resource ratio by collecting our best software in clean modular structures into our library

PORT, having been carefully designed to be an instrument for program development, provides a good foundation and framework for this work

ACKNOWLEDGEMENT

The members of the advisory board of the PORT Library are W Stanley Brown and Andrew D Hall; the editor is P A Fox, and the board of editors has as members, James L Blue, John M Chambers, Linda C Kaufman, Norman L Schryer and Daniel D. Warner The work all these people have put into PORT is beyond measure Thanks must also be given to Stu Feldman for providing the language, EFL, and to our users who make it all worthwhile

REFERENCES

1. P. A. Fox, A D. Hall, and N L Schryer, "The PORT Mathematical Subroutine Library." ACM Transactions on Mathematical Software, Vol 4, No. 2, June 1978, pp. 104-126

2 P A Fox, A D Hall, and N L Schryer, "ALGORITHM 528, Framework for a Portable Library [Z]," ACM Transactions on Mathematical Software, Vol 4, No 2, pp. 177-188

3 P. A. Fox, PORT - Users Manual, Bell Laboratories, 1977.

4. B G. Ryder, "The PFORT verifier," Software - Practice and Experience, Vol 4(1974), pp 359-377

D16

N79-12731

ORIGINAL PAGE IS OF POOR QUALITY

THE IMSL ENVIRONMENT FOR SOFTWARE DEVELOPMENT

T. J. Aird and D. G. Kainer

IMSL
Houston, Texas

ABSTRACT

IMSL has developed a set of macros and a file naming convention that automates the subroutine development and testing process over ten computer types. The IMSL software development system is implemented on a Data General Eclipse C330 computer with 256K bytes of central memory and 192M bytes of disk storage using the AOS Operating System. RJE activity is handled by a Data 100 communications computer. The system allows the programmer to work with basis decks. Distribution decks are generated, by the IMSL Fortran converter, as they are needed for testing and whenever the basis deck has been modified.

THE IMSL LIBRARY consists of approximately 450 Fortran subroutines in the areas of mathematics and statistics. At this time IMSL serves over 650 subscribers. The library is available on the computers of ten manufacturers as follows:

IBM 360/370 series
Xerox Sigma series
Data General Eclipse series
Digital Equipment series 11
Hewlett Packard 3000 series
Univac 1100 series
Honeywell 6000 series
Digital Equipment series 10 & 20
Burroughs 6700/7700 series
Control Data 6000/7000 and
Cyber 70/120 series

Each subroutine has an associated minimal test routine which executes the manual example of usage and an exhaustive test program designed to exercise the various types of usage for each routine and the range of input data which might be employed. This amounts to over 9,000 individual programs which IMSL must produce and support (subroutines and minimal tests). These programs, together with the exhaustive test programs for each subroutine, constitute over 10,000 total programs which must be managed efficiently. In addition, each one of these subroutines must be tested on at least one of each of the ten computers.

At best, the production and support of the library is a very complicated task. But IMSL has developed a set of macros and a file naming convention that automates the subroutine development and testing process on these ten computer types. The IMSL software development system is implemented on a Data General Eclipse C330 computer with 256K bytes of central memory and 192M bytes of disk storage using the AOS Operating System. This machine is also used for the testing of the Data General Library. The testing on the nine other supported machines is done by remote job entry via a Data 100 computer which can be programmed to emulate the environments necessary for job entry to the various computer types.

The file naming convention used by IMSL is designed to clearly distinguish the different types of files used in the development of the library.

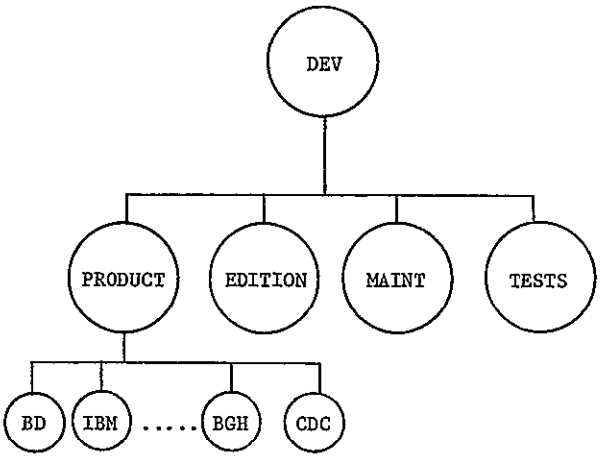
The naming convention consists of the following:

name ≡ <IMSL routine name. 6 characters or less>
prog ≡ <IMSL test program or associated subprogram name. 6 characters or less>
computer ≡ <IBM|XEROX|DGC|DEC11|UNIVAC|HIS|DEC10|BGH|CDC|HP3000|H32|H36|H48|H60|ALL>
ALL => portability across entire computer set.
H32=> portability across H32 computer set (etc. for H36, H48, and H60). The H32 computer set consists of IBM, XEROX, DGC, DEC11 and HP3000. The H36 set consists of UNIVAC, HIS and DEC10. The H48 set consists of BGH. The H60 set consists of CDC.

File Name	Description
name.BD	Basis deck for IMSL routine "name"
name.computer	Distribution deck source file
name.DGC.OB	DGC object files
name.ET PR name.MT.PR	DGC program files
RJE.name.ET.computer RJE.name.MT.computer	RJE jobfile as produced by the system
name.ET.DGC.LISTFILE name.MT.DGC.LISTFILE	Output listfile as produced by the system for DGC
name.ET.prog.BD name.ET.prog.computer	Exhaustive test program or subprogram
name.ET.LL computer	Required programs and routines list for the exhaustive test
name.ET.DD.computer	Data for exhaustive test
name.MT.prog.BD name.MT.prog.computer	Minimal test program or subprogram
name.MT.LL.computer	Required programs or routines list for the minimal test
name.MT.DD.computer	Data for minimal test

This naming convention gives a unique name to each file which enables identification of the file type by the programmer and the system.

The development system makes use of the Data General convention of logically splitting disk space into independent sections called directories. Each programmer has his/her own directory called a working directory where all modifications to IMSL programs take place. Modification is not permitted elsewhere. The main directory in the development system is called DEV. This directory contains all of the programs used by the system as well as the subdirectories used. There are four subdirectories of DEV. Each of these directories is write-protected to insure that someone cannot unknowingly alter their contents. The first is called PRODUCT. This directory contains eleven subdirectories, one for each of the computer types and one for the basis decks. Each of these subdirectories contains the source distribution files for each of the distributed programs (subroutines and minimal tests) for that computer. The DGC subdirectory also contains the compiled object files of the Data General product set. The second subdirectory in DEV is called EDITION. This subdirectory contains all of the files for new programs and modified current programs which will be released in the next edition of the library. The third subdirectory of DEV is called MAINT. This subdirectory contains the files for the versions of the programs which have been modified and are scheduled for release at the next maintenance of the library. The final subdirectory is called TESTS. This directory contains the source files for all versions of the exhaustive test programs, their associated subroutines, and the data. It also contains the required programs and routine lists for the exhaustive tests and the minimal tests.



At the heart of the development system is the Fortran Converter. This is a program which performs automatic conversion of Fortran programs and subprograms from one computer-compiler environment to another. This conversion is done automatically via built-in features of the converter and specifically via converter instructions inserted in the program by the programmer. Thus the code for all versions (computer types) of a program are contained in one file called a basis deck. The programmer makes modifications to the basis deck only. Distribution decks are generated by the converter as they are needed for testing and whenever the basis deck has been modified. This general approach to handling portability problems is explained in (1) and (2).

The programmer works at a CRT console and uses the development system by executing the command RUN.DEV When the target computer is Data General, this command will run a job including all necessary basis deck conversions, compilations, binding, and execution of the program file. For all other computers, a job will be set up (with appropriate job control language) for running at a remote site and submitted to the appropriate remote job entry site queue. The command is issued in the following way:

```
RUN.DEV/C=computer[/other optional
switches] name.ET (or name.MT)
```

where computer is one of the ten distribution environments and name is the name of the IMSL routine to be tested. The development system begins the execution of the command by locating the appropriate list of the required routines contained in an LL file. The system then determines which (if any) basis decks located in the programmers working directory must be converted by examining the time last modified. If the time last modified of a basis deck is later than the time last modified of its corresponding distribution file, conversion must take place. Next, for all computers other than Data General, a job is built consisting of the routines required for running and the appropriate job control language. For every computer, there is a default site for running. Unless specified otherwise by the programmer via a /COMPILER=type switch, the default job control language is used. The JCL is contained in files named

```
computer.compiler.cc
```

These files contain the commands necessary for compilation, binding, and execution of a Fortran job at the job site After the job is built, it is submitted (unless otherwise specified) to the default site queue for that particular computer or to a specific queue via the /QUEUE=site switch From here, an operator copies the job to magnetic tape and places the tape on the Data 100 for input to the remote site. For Data General, after conversion, the required source files are compiled, if necessary. This determination is made in the same manner as the determination for conversion. This is followed by binding and execution of the program file. The programmer is informed upon completion of the job (execution for Data General, queue submission for other computers). The system monitors itself during the execution of a RUN.DEV command and, if at any time during the execution an error occurs, the system will terminate execution of the command and send the user an appropriate error message. In addition, by specifying the /D switch, the system will place certain vital communications files in the working directory to help the user in debugging his problem.

The system will perform conversions, compiles, binds, and execution only in the working directory (that is, the directory from which the programmer gave the command). If a basis deck or its corresponding distribution file is not found in the working directory, the system will try to locate the file by searching the TEST directory, the

MAINT directory, the EDITION directory, and finally the appropriate directory of PRODUCT. This enables the programmer to run a test without necessarily having all of the required routines in the working directory. The programmer can alter which directories are searched by specifying the /SL switch and setting up his own list of directories. In addition, the programmer can specify the /CURRENT switch which negates the search of MAINT and EDITION. This is particularly useful if the programmer is trying to verify a problem which occurred in the current release of the library because it uses the version of the library which is out in the field at that time.

Both the Fortran converter and the development system are written in modular form to insure easy readability and debugging of the code. The system itself was designed in such a way as to easily facilitate the addition of a new computer into the product set. It also allows the programmer unlimited flexibility in varying his/her specific runs. The command generally requires 30 to 60 seconds of CPU time and takes between 15 and 20 minutes of elapsed time to execute to completion. Therefore, the programmer usually issues the command from one of four batch streams, freeing his/her terminal for other uses.

The production and support of a multi-environment subroutine library is a complicated task, requiring many tedious details. The IMSL development system makes life much easier for the programmers undertaking this task. Through the issuance of a simple, one line command, the programmer can cause the execution of a job, freeing him/her from annoying but necessary details such as obtaining the right version of a required code or unintentionally modifying a code. This enables the programmer to concentrate on the task at hand: supporting and enhancing a state-of-the-art subroutine library.

Other software tools available through the IMSL software development system are listed below:

Name	Purpose
HELP	Provide on-line documentation for all system commands
RUN.CVT	Execute the Fortran converter to produce a specific distribution deck
RJE	Submit a job for execution at one of the RJE sites
RJE.S	Status of RJE activity
MAGCARD	Send computer readable documentation to an IBM mag card typewriter
CLEAN	Reformat source code according to IMSL conventions - runs on a CDC computer
REVIEW	Review a code and list deviations from IMSL conventions
PFORT	Submit a code to be run through the PFORT verifier - runs on an IBM computer
BRNANL	Submit a code to be run through the branch analysis program - runs on a CDC computer
SEQUENCE	Sequence a deck
STRIP	Remove sequence numbers and trailing blanks from a deck
SPLIT	Split a file into separate program units
COMPARE	Compare two codes and list differences
EDIT	Edit a file via Data General's text editor SPEED

REFERENCES

1. T. J. Aird, "The IMSL Fortran Converter: An Approach to Solving Portability Problems". New York, Springer-Verlag, "Portability of Numerical Software", Editor, Wayne Cowell, 1977, pp. 368-388

2. T. J. Aird, E. L. Battiste, and W. C. Gregory, "Portability of Mathematical Software Coded in Fortran", "ACM TOMS", Vol. 3, No. 2, June, 1977, pp. 113-127.

3. "Advanced Operating System (AOS), Command Line Interpreter, User's Manual", Data General Publication 093-000122, 1977.

Transportability in Practice - Recent Experience with the NAG Library

J.J. Du Croz

Numerical Algorithms Group Limited
7 Banbury Road
Oxford OX2 6NN
England

ABSTRACT

Two guiding principles for the development of the NAG Library are:

a) the algorithms must be adaptable to the characteristic of the computer on which they are being run;

b) the software must be transportable

(These concepts have been discussed in detail elsewhere.)

The purpose of this talk is to discuss how these principles have stood the test of practice over the past two years and how NAG's approach to library development has been refined in the light of experience.

The adaptability of algorithms is achieved with the aid of machine-dependent parameters, available through calls of library functions. The initial set of parameters was concerned only with the arithmetic properties of the computer. A new parameter is now being introduced, concerned with storage organizations: it will enable algorithms for certain operations of linear algebra to adapt, when run on a paged machine, to the approximate amount of real store available, and hence to achieve a dramatic reduction in page-thrashing.

The transportability of the software is proven by the fact that the NAG Library has now been implemented on 25 different computer systems - including several minicomputers, which required no fundamental change of approach. The talk will discuss various interesting problems which have arisen in the course of implementation. Frequently it has been deficiencies in the compiler and associated software that have caused the problems, rather than defects in the NAG routines. NAG's test software (though not always ideally suited to the task) has proved remarkably effective at detecting these problems before the Library is released to sites. This justifies NAG's policy of always distributing a tested compiled library (in addition to the source-text).

D17

N79-12732

PORTABILITY AND THE NATIONAL ENERGY SOFTWARE CENTER

Margaret K. Butler, National Energy Software Center

Argonne National Laboratory
Argonne, Illinois, U.S.A.

ABSTRACT

The software portability problem is examined from the viewpoint of experience gained in the operation of a software exchange and information center. First, the factors contributing to the program interchange to date are identified, then major problem areas remaining are noted. The import of the development of programming language and documentation standards is noted, and the program packaging procedures and dissemination practices employed by the Center to facilitate successful software transport are described. Organization, or installation, dependencies of the computing environment, often hidden from the program author, and data interchange complexities are seen as today's primary issues with dedicated processors and network communications offering an alternative solution.

THE NATIONAL ENERGY SOFTWARE CENTER (NESC) is the successor to the Argonne Code Center, originally established in 1960 to serve as a software exchange and information center for U S. Atomic Energy Commission developed nuclear reactor codes. The Code Center program was broadened to an agency-wide program in 1972, and the scope of the activity expanded further with the organization of the Energy Research and Development Administration (ERDA) and then the Department of Energy (DOE). The Center's goal is to provide the means for sharing of software among agency contractors, and for transferring computing applications and technology developed by the agency to the information-processing community.

To achieve these objectives the NESC:

1. Collects, packages, maintains, and distributes a library of computer programs developed in DOE research and technology programs.
2. Prepares and publishes abstracts describing the NESC collection.
3. Checks library contributions for completeness and executes sample problems to validate their operation in another environment.
4. Consults with users on the availability of software and assists them in implementing and using library software.
5. Compiles and publishes summaries of ongoing software development efforts and other agency-sponsored software not included in the collection because it is felt to be of limited interest.
6. Maintains communications and exchange arrangements with other U.S. and foreign software centers.
7. Coordinates acquisition of non-government software for the Department of Energy.
8. Initiates and participates in the development of practices and standards for effective interchange and utilization of software.

About 850 computer programs covering subject classifications such as mathematical and computer system routines; radiological safety, hazard, and accident analysis; data management; environmental and earth sciences, and cost analysis and power plant economics make up the current collection. Each year over 1000 copies of library software packages or authorizations for their use are disseminated in response to requests from DOE offices and contractors, other government agencies, universities, and commercial and industrial organizations. Over 500 organizations are registered as participants in the program. Of these 333 are registered as contractors, or under exchange arrangements. The remainder pay the cost of the materials and services requested.

It is clear from the enumeration of NESC activities that the major portion of the Center's program is devoted to the task of making it possible for individuals, other than the program author, to use the computer program in their own computing environment, which is different from the environment in which the author developed the software. The Center's dissemination statistics attest to a measure of success in the task. This success is due in part to the DOE sponsors of program development projects who, recognizing the need for software sharing, have encouraged authors to attempt to develop portable software, and to the authors who, self-motivated, entered the initial development stage with the avowed intent of producing a transferrable product. But success must be attributed in large part, too, to the availability of programming language and program documentation standards, and to the program review and evaluation procedures and dissemination practices established by the Center.

STANDARDS

In 1964 when the first FORTRAN language standards were published, program developers welcomed them enthusiastically as the means by which programs, developed for today's environment, could be readily transferred to tomorrow's, or moved to someone else's, quickly and with minimal cost. Installations attempting to achieve this promised portability, however, were frustrated by the variety of implementations produced by the compiler writers, each claiming conformity with the standard, but offering, in addition, special enhancements exploiting their particular hardware.

At the time a standards committee of the American Nuclear Society sent a letter to the Editor of the Communications of the Association for Computing Machinery urging the computing community to exert pressure on the compiler developers to implement the standard, and where deviations existed to flag

accepted non-standard statements and standard statements implemented in a non-standard fashion, describing these variations in the compiler documentation. The computing community not only failed to endorse this early plea for a standard that could be used to achieve software portability, but the Editor held up publication of the letter for six months because FORTRAN was mentioned explicitly and the ACM might appear to be showing a preference for FORTRAN over other programming languages

While programming language standards have not proved to be the ready remedy to the portability problem first envisioned, they have provided the necessary first step Authors pursuing the goal of producing portable software have, by restricting themselves to a "portable" subset of the standard language, common to nearly all compilers, been able to produce an easily transferrable product. Program verification tools, such as PFORT, have proved helpful in this activity.

Over the past decade the American Nuclear Society's ANS-10 standards committee has produced a series of standards to assist authors and developers of scientific and engineering computer programs in preparing software to be used by colleagues at other installations. These include ANSI N413-1974 entitled "Guidelines for the Documentation of Digital Computer Programs" and ANS-STD.3-1971, "Recommended Programming Practices to Facilitate the Interchange of Digital Computer Programs", both of which were adopted by the AEC's Reactor Physics Branch, along with the ANSI FORTRAN standard, for its program development projects. Recently, this committee completed another guidelines document, this one titled "Guidelines for Considering User Needs in Computer Program Development". It is presently under review by the parent committee

PACKAGING PROCEDURES

The Center's software package is defined as the aggregate of all elements required for use of the software by another organization, or its implementation in a different computing environment It is intended to include all material, associated with a computer program, necessary for its modification and effective use by individuals other than the author, on a computer system different from the one on which it was developed. The package is made up of two basic components: the computer-media material and the traditional printed material or hard-copy documentation. The material may include all of the following:

1. Source decks: source language program decks or card-image records.
2. Sample problems test case input and output for use in checking out installation of the software
3. Operating system control information: operating system control language records required for successful compilation of the software and execution of the sample problems in the author's computing environment. This element includes device assignment and storage allocation information, overlay structure definitions, etc.
4. Run decks: object decks or load modules prepared by the language processors preliminary to execution of the software. This element is redundant when included in conjunction with the equivalent source decks for interchange between users of like systems
5. Data libraries external data files required for program operation, e.g., cross section libraries, steam table data, material properties. To provide for exchange across machine-lines a decimal or alphanumeric form is recommended. Routines for transforming the data to the more efficient binary or machine-dependent representation should be included as auxiliary routines.
6. Auxiliary routines or information supplementary programs developed for use in conjuction with the packaged software; e.g., to prepare or transform input data, to process or plot program results, edit and maintain associated data libraries.
7. Documentation: traditional reference material associated with the development of the software and its application If documentation is provided in machine-readable form it is classified as auxiliary information. This item may be a single report or several independent documents.

Not all seven elements are required for every software package; however, items 1,2,3, and 7 are rarely absent from scientific and engineering applications which make up the bulk of our library.

In our review process the package elements are examined for consistency and completeness. Whenever possible, source decks are compiled and test cases and run decks executed to ensure the output generated in another environment agrees with that produced at the developer's installation. This evaluation process provides a good check also, of the adequacy of the documentation elements. If the submitted documentation proves inadequate for our staff to evaluate the software additional information is sought and incorporated in the package documentation. Frequently an NESC Note is written for this purpose. If data libraries are present an effort is made to include these in machine-independent form. This is especially important for the first, or original, version of a program. When conversions to other computer systems, i.e. additional machine versions, are considered, machine dependencies reflecting significant convenience to the users of that system are accepted. An attempt is made to retain in the collection one version of each library program in a form amenable to transfer.

Special software tools have been developed to verify that all routines called are included in the package, to perform rudimentary checks for uninitialized or multiply-defined variables, unreferenced statement numbers, active variables in common and equivalence blocks together with block names and addresses, etc., and to convert between a variety of character codes such as IBM EBCDIC, CDC Display Code, and UNIVAC Fieldata.

DISSEMINATION PRACTICES

The computer-media portion of the program package is generally transmitted on magnetic tape, however, card decks will be supplied for card-image material upon request The tape recording format to be used in filling a request can be specified by the potential user to suit his computing environment, and, whenever possible, the Center will provide the format, character code, density, and blocking requested.

The Center maintains records of all package transmittals and, should an error be detected or a new edition be received, all recipients of the affected program package are notified.

PROBLEMS

A large number of problems encountered today in software sharing are traceable to in-house modifications of vendor-supplied systems, locally-developed libraries and utility routines, and installation conventions Recognizing the problem of reproducing the performance of a program at another installation independent of the "local" system, the ANS-10 standards committee introduced the concept of an installation-environment report in its "Code of Good Practices for Documentation of Digital Computer Programs", ANS-STD.2-1967. The idea was that this report would document those timing, plotting, special function, and other local system routines which would have to be transferred with locally-developed software, or replaced with their counterparts at another site before the software could be successfully executed. It was suggested that each program-development installation package their collection of the documented routines as a library package so that users of, for example, the XYZ Laboratory's software would be able to create the necessary XYZ environment either by implementing these environmental routines or with acceptable alternative routines. Several packages in the Center collection are of this nature; the Bettis Environmental Library and the NRTS Environmental Routines are two of them.

Most computer centers, however, have never attempted to define such a package Program developers are frequently unaware of the routines automatically supplied by the local system and seldom informed when changes or modifications to such routines are effected.

Proprietary software can magnify installation dependency problems, and programs utilizing graphical output are always a challenge to exchange. In most cases the Center, and each receiving organization, is required to insert the equivalent local plotting routines before the test cases can be run Even when organizations have the same commercial software product, they will probably have different plotting devices, and, if not, you can bet each location chose to implement its own unique enhancements--after all, graphics is an art!

One-of-a-kind compilers and parochial operating systems used at some installations have proved to be a significant deterrent to program interchange. Of growing concern with the increasing use of database systems is data exchange. A DOE Interlaboratory Working Group has committed its members to working with the ANSI Technical Committee X3L5 to develop specifications for an Information Interchange Data Descriptive File.

ARPANET and the Magnetic Fusion Energy Computer Center offer a different solution to the software portability problem. Instead of developing portable software and making it generally available such facilities encourage the development of centers of excellence which provide and maintain the latest and best software for a particular application on a dedicated processor, accessible to the user community. That is the other end of the spectrum.

A TRANSPORTABLE SYSTEM FOR MANAGEMENT AND EXCHANGE OF PROGRAMS AND OTHER TEXT

W. V. Snyder

Jet Propulsion Laboratory
4800 Oak Grove Drive
Pasadena, California 91103

ABSTRACT

Computer software is usually exchanged between different computer facilities via punched cards or magnetic tape. For up to about 1000 images, cards are cheaper and probably easier to deal with than tape. The primary problem with cards is the variety of punch codes. Frequently, one also has the minor nuisance of repunching cards damaged in transit. For larger amounts of data, tape is cheaper, but there are so many tape formats in use that the recipient frequently has trouble reading a tape, even if the format is simple and well defined. Occasionally, one has the problem of parity errors, which make a tape essentially worthless. When test data, modules in several languages, computer output or documentation are included, the lack of organization in the material can cause a substantial amount of unnecessary human labor.

This paper presents a system for exchanging information on tape, that allows information about the data to be included with the data. The system is designed for portability, but requires a few simple machine dependent modules. These modules are available for a variety of machines, and a bootstrapping procedure is provided. The system allows content selected reading of the tape, and a simple text editing facility is provided. Although the system recognizes 30 commands, information may be extracted from the tape by using as few as three commands. In addition to its use for information exchange, we expect the system to find use in maintaining large libraries of text.

THE MOTIVE FOR DEVELOPING THIS PROGRAM

was the experience of receiving tapes from many correspondents. We dealt with most correspondents only once or twice. We received tapes written in every possible density, both parity modes, several character codes, and having a variety of block and record lengths. We see three solutions to this problem. Most computer centers have access to a program that can handle fixed length records, written in fixed length blocks, using a popular code such as ASCII or EBCDIC. When the characteristics of the medium were correctly provided, we had good success with a program of this type*. Unfortunately, this information was not always provided, and was sometimes incorrect. Another solution is for some organization to promulgate a standard for record lengths, block lengths, codes and parity modes. Then if such information is not provided, the standard is a reasonable guess. Neither approach can cope with disorganization of the data, or with parity errors. We chose therefore to write a transportable program to enforce a standard recording format, organize the data and provide for error recovery. This relieves the sender of the responsibility for sending information about character codes, record lengths and block lengths with the tape. He must, of course, still tell the receiver the tape density, and whether it is a seven- or nine-track tape. Since some binary numeric information is recorded, only odd parity tapes may be used with this program.

*We used two programs, known as BLOCK and UNBLOCK, written in Univac-1100 assembler language at the University of Maryland Computer Science Center.

RECORDING FORMAT

Most computer systems can deal with ASCII information in a natural way. In order to use nine-track tape conveniently, we represent the seven-bit ASCII code using eight bits, with the high-order bit zero. The program does not, however, enforce this convention rigidly. Certain information must be encoded in this way, but the textual information may be encoded in any way that may be represented by a string of eight-bit units. It is preferable that all information be encoded in some standard form, and we hope that all implementations of the program will use ASCII code for the textual information.

Some computers can read or write tapes containing blocks consisting of an integral number of words, and can read tape blocks of arbitrary length only with difficulty. For example, a tape containing blocks consisting of ten 80-character records could be read only with difficulty on a Univac-1100, which expects nine-track tapes to contain blocks consisting of a multiple of nine characters, and could not be written on a Univac-1100. We therefore selected a block size having factors of nine (for 36-bit words), fifteen (for 60-bit words) and four (for 32-bit words). These factors also guarantee that the block will be an integral number of words if it is written on a seven-track tape. The program uses data the same for seven- and nine-track tapes.

Since information may be recorded on magnetic tape in blocks of arbitrary length, separated by gaps of fixed length, one can use less space on the tape to record a given amount of data by writing longer, and therefore fewer blocks. We chose to write information in blocks of 7200 characters. This block size allows efficient use of space on tape, and usually fits into a minicomputer memory. A 180-character label is the first block written on every tape. Information in the label includes the block size. If the program does not fit in available memory, smaller blocks may be written. The program can read the smaller blocks automatically. This was required in one minicomputer

implementation of the program. We recommend that all implementations retain the capability to read 7200 character blocks. Further conservation of space on the tape is achieved by compressing the data. To compress the data, consecutive occurrences of blanks (or another character if desired) are removed, and replaced with an encoded representation requiring less space. A compressed Fortran program usually occupies about one third the space otherwise required.

DATA MANAGEMENT FACILITY

Although the problem of dealing with variable and frequently uncertain physical characteristics of the transmission medium was irritating, the problem that consumed most of our time was the uniform lack of organization of the information on the tape. We received programs in several languages, subprograms with several test drivers, multiple versions of a program, test data, computer output and documentation, with no indication of what was to be done with the information. In such situations, much effort was spent organizing the information before it could be used. We therefore developed not only a program to read and write tape, but also a transportable data management system for textual information.

Our data management scheme consists of recording each program, subprogram, listing or data group as a separate module of text. Helpful information about the module is recorded with the module. The minimum information required with each module is a name. For more complete identification of the module, one may record the data type (language for modules that are programs), machine type, authors' names and addresses, and bibliographic references. To facilitate management of programs consisting of several modules, one may record the names of groups of which the module is a member, and keywords related to the module. To control changes to modules, a simple but flexible updating mechanism is provided, and the updating history is recorded. To record information that does not fall into any of the specified categories, one may include comments. We call this information control information. All control information is recorded with the text of the module. The text and control information can be examined and updated separately, but they remain together on the tape.

A data management system requires a command language. In specifying the command language for the Exchange Program, our goals were simplicity and comprehensive flexibility. The use of the program is oriented primarily toward the receiver of the tape. Although the program acts on 30 commands, information may be extracted from the tape with as few as three commands:

INTAPE = Fortran unit number of input tape
OUTPUT = Fortran unit number of native format file
COPY = List of module numbers.

To create a new tape requires, at a minimum, the following commands:

TITLE = Title of tape
SITE = Site at which the tape is being written
OUTAPE = Fortran unit number of the tape
DATE = Date written (YYMMDD) [May be provided automatically.]

Each module of the text must then be preceeded by

INSERT = Name of module
TEXT

and followed by an end of text signal. If more information about the module than its name is to be provided, more commands are required.

ERROR DETECTION AND CORRECTION

The program currently contains two error detection mechanisms. First, it uses the error detection mechanism of the supporting operating system. Second, it records a sequence number in each block, and checks it during reading. It also records in each block the location of the first record that starts in the block, the number of the text module of which it is a member, and the location of the first record of the first text module that begins in the block, if any. We plan to use this information for partial error recovery. We also plan a more ambitious error control algorithm, capable of detecting and correcting up to 72 consecutive erroneous characters, at up to four different places in each block. It can be implemented in a transportable way, requiring only a machine sensitive exclusive-or primitive operation. For the 7200 character block chosen as the standard for the Exchange Program, only 113 characters of error control information are required. The design of the block format includes provision for this information.

EXPERIENCE

The program has been used at JPL to manage the collection of algorithms submitted to ACM TOMS, for weekly exchange of data between a DEC PDP-11/55 and a Univac-1108, and occasional exchange of data between a Univac 1108, Sperry (formerly Varian) 72, and a DEC PDP-11/55. The program was used to transmit the JPL mathematics library to a DEC PDP-10 at the California Institute of Technology, and is currently used there to retrieve modules of the JPL mathematics library from the exchange tape. It was also used to transmit information to a CDC-6600 at Sandia Laboratories. Experience in implementing the program on the DEC PDP-11/55 and on the DEC PDP-10 indicated that changes in the interface between the portable and non-portable parts of the program are desirable. In particular, the DEC Fortran environment requires that data files be explicitly opened (with a non-portable statement) before they are used. Thus, a subprogram thought to be portable does not work on DEC machines. We expect to change the interface between the portable and non-portable parts of the program to concentrate potentially non-portable requirements in fewer places. When we make that change, we will also add a few commands.

SUMMARY

We have developed a transportable program for exchange of textual information that provides several advantages over previous methods. The program enforces the use of a standard tape format, uses tape efficiently, organizes the information on the tape, provides for simple retrieval of information from the tape, and provides for error recovery. Since the program is transportable, it is used similarly on all computer systems. Thus, once one learns to use the program, one may use the program on many computer systems with little additional effort.

ACKNOWLEDGEMENTS

The author thanks the following (in alphabetical order) for their help: Julian Gomez of JPL implemented the program on a DEC PDP-11 using RSX-11. Jeff Greif at the California Institute of Technology implemented the program on a DEC PDP-10. Dick Hanson and Karen Haskell at Sandia Laboratories in Albuquerque wrote some of the IBM system 360 modules. Karen Haskell implemented the program on a CDC-6600. Fred Krogh of JPL provided most of the requirements and several ideas for the command language. Robert McEliece of JPL provided an error detection code. West Coast University in Los Angeles provided free computer time to implement the program on a Varian 72.

This paper presents the results of one phase of research carried out at the Jet Propulsion Laboratory, California Institute of Technology, under contract No. NAS 7-100, sponsored by the National Aeronautics and Space Administration.

Dj9

N79-12734

Using Pascal for Numerical Analysis

by

Dennis Volper (speaker)

Institute for Information Systems, C=021
University of California, San Diego
La Jolla, CA 92093

and

Terrence C. Miller

Computer Science Segment
Applied Physics & Information Sciences Dept., C-014
University of California, San Diego
La Jolla, CA 92093

ABSTRACT

The data structures and control structures of Pascal enhance the coding ability of the programmer. Recent proposed extensions to the language further increase its usefulness in writting numeric programs and support packages for numeric programs.

PASCAL HAS THE ADVANTAGE of being a highly structured language. In addition, it was specifically designed "to make available a language suitable to teach programming as a systematic discipline based on certain fundamental concepts clearly and naturally reflected by the language."[2]* It has been noted by R. Block [1] and others that structured approaches to algorithms reduce programming time. It has been the experience of the UCSD Pascal project that students quickly learn the structures of the language and are rapidly able to maintain and modify programs of considerable complexity. For example the code in the UCSD Pascal system is the work of student programmers. The majority of those programmers are undergraduates. Maintainability and verification of programs are made easier in Pascal because the structure of the program closely resembles the structure of the algorithm it represents.

The popularity of the language Pascal is growing. Work is progressing to remove remaining deficiencies in the language. The Institute for Information Systems at UCSD [4] has developed and is continuing to improve a machine independent Pascal system for small computers. Also this summer the Workshop on Extensions to Pascal for Systems Programming [3] has recommended extensions which will enhance the applications of Pascal to numerical analysis. Parallel to the work of the workshop, an international group of Pascal experts is attempting to construct a standard for Pascal and its extensions.

PROGRAMMING IN PASCAL

In this section we will describe certain features of Pascal which are useful in writting support packages for numerical programs as well as the numerical programs themselves.

DATA STRUCTURES IN PASCAL - Data handling is simple and efficient because the Pascal Language supports the declaration of data structures. The programmer may use the base types of the language to build structured types and may even create files of user declared types. These complex types may be manipulated, either as units or by their various parts. For example to exchange two rows of a matrix, rows can be treated as single entities.

```
TYPE ROW=ARRAY[0..6] OF INTEGER;
     MATRIX=ARRAY[0..5] OF ROW;
VAR A:MATRIX;
    R:ROW;
BEGIN
  R:=A[1];
  A[1]:=A[2];
  A[2]:=R;
END;
```

This reduces the details of coding required to do data handling and simultaneously reduces the possibility of programmer error. Data structures can be declared to be PACKED causing the maximum number of variables to be in each word. This provides considerable space savings for any non-numeric data which must be associated with numeric computations.

To further enhance the potential space savings variant records are allowed. Dynamic variables may be variant records, in which case only the space necessary for the variant declared is needed. Strong type checking is maintained as a tool to validate data handling and thereby minimize programming errors.

*Numbers in parentheses designate references at end of paper

Thus the language is extremely convenient as a data handler, with a simplicity exceeding COBOL and a flexibility exceeding FORTRAN and ALGOL. The availability and ease of using files make it an excellent language to use as a preprocessor for numerical data. The compactness of its representations has the consequence of requiring smaller amounts of source code.

Dynamic variables called pointers permit the explicit expression of linked structures such as sparse matrices.

```
TYPE ARRAYENTRY=
          RECORD
            DATA:REAL;
            ROWLINK:^ARRAYENTRY;
            COLUMNLINK:^ARRAYENTRY
          END;
```

Note that these structures, as well as arrays of any data structure, have the advantage that pieces of logically grouped information are represented as components of a logical structure, rather than as entries in parallel structures.

CONTROL STRUCTURES - Pascal is a highly structured, recursive language with a design which encourages top-down programming. With its 3 looping (WHILE,REPEAT-UNTIL,FOR) 2 branching (CASE,IF-THEN-ELSE) there are a limited number of control constructs to understand, yet they are sufficient to express any algorithm. In large numerical programs a "90-10" rule appears to hold. Particularly in the interactive environment, the bulk of the source code represents the control structures and user interface of the program, with intense numeric calculations representing a small fraction source code. The block structure, the simplicity of the Input and Output commands make these easy to code.

Syntax has been specified for type secure compilation of external modules and their inclusion into Pascal programs.[4] In addition UCSD Pascal permits the inclusion of procedures from native code modules into Pascal programs. This will permit the writing of small pieces of highly used code in assembly language thereby increasing speed. It will also permit linking user written data gathering, or hardware monitoring routines into Pascal programs.

MACHINE INDEPENDENCE - One of the goals of Pascal is machine independence. By adhering to standard Pascal it should be possible to compile and run a program on any implementation. Validation suites are being made available for the purpose of comparing a given implementation with the standard. UCSD Pascal is designing machine independence into its numerical calculations by converting its numerical package to the proposed IEEE standard. Using this standard it is possible to gain machine independent results to floating-point calculations, that is, the same calculation performed on two machines (of equal word length) will produce bit equivalent results.

INTERPRETIVE ADVANTAGES - Interpretive implementation, packing of data, and compactness of source code combine to allow a larger portion of the available memory to be allocated to data storage.

NUMERICAL PROGRAMMING IN PASCAL

This section provides a description of those features of Pascal which are particularly relevent to numerical analysis. We will be describing both features which affect only program style as well as those which can cause significant execution differences between FORTRAN and Pascal implementations of the same algorithm.

ARBITRARY ARRAY INDICES - Let us assume an algorithm which works on 3 vectors in synchronization. A FORTRAN programmer could declare 3 vectors:

```
INTEGER RED(10),GREEN(10),BLUE(10)
```

or if he was worried about page faults and confident that his installation's FORTRAN stores arrays by columns he would declare:

```
INTEGER COLORS(3,10)
```

totally obscuring the separate identities of the three vectors.

In contrast a Pascal programmer faced with the same problem could declare:

```
TYPE COLOR = (RED,GREEN,BLUE);
VAR  COLORS: ARRAY [1..10] OF ARRAY [COLOR]
     OF INTEGER;
```

using array index constants which preserve readability without sacrificing the ability to loop through the colors since:

```
FOR C:=RED to BLUE DO ......
```

is a legal Pascal loop statement.

NUMERICAL ARRAY INDICES - The Pascal programmer also has the freedom to use any desired lower limit for numeric (integer) array indices. Unfortunately, it is not possible to have arbitrary spacing of indices.

ADJUSTABLE ARRAY PARAMETERS - The current definition of Pascal allows a procedure which takes arrays as arguments to be called only with actual arguments of one fixed size. That restriction has, however, been recognized as a mistake, and allowing procedure headings of the form shown in the example given below has been proposed [3]:

```
PROCEDURE P(A: ARRAY [LOW..HIGH:INTEGER]
               OF INTEGER)
```

When procedure P is called, the local variables LOW and HIGH will be automatically assigned the index limits of the array used as the actual argument. Passing the array size as a separate argument to the function as is done in FORTRAN will not be required.

COMPLEX ARITHMETIC - The current definition of Pascal does not provide for complex arithmetic. However, adding that

facility has been proposed [3]. The proposed extension includes a predefined type COMPLEX, the functions necessary to take complex numbers apart (RE, IM ,ARG), and a way of creating complex constants and values. The standard mathematical functions will also be extended to handle complex numbers.

LONG INTEGERS - The UCSD Pascal system now provides the capability of declaring the minimum number of digits that must be contained in each integer number declared. All arithmetic operations involving large range numbers can generate intermediate results of at least the specified size without causing overflow.

SHORT INTEGERS - The Pascal sub-range feature allows the user to declare that a given variable can have only a very small range of values. This can lead to considerable savings in storage as shown in the following example:

```
VAR PACKED ARRAY [1..16] OF 0..3
```

which uses only 2 bits for each array element.

REAL NUMBERS - Pascal currently defines one floating point precision, and that is machine dependent. However, the UCSD Pascal system will incorporate the IEEE floating point standard as soon as it is finalized (at least for 32 and 64 bits).

REFERENCES

[1] R.K.E. Block, "Effects of Modern Programming Practices on Software Developement Costs", IEEE COMPCON Fall 1977.

[2] K. Jensen and N. Wirth, "Pascal User Manual and Report" Springer-Verlag, 1978.

[3] T.C. Miller and G. Ackland ed., "Proceeding of the Workshop on Systems Programming Extensions to Pascal", Institute for Information Systems, Univ. of California, San Diego, July 1978.

[4] K.A. Shillington, "UCSD (Mini-Micro Computer) Pascal Release Version I.5 September 1978", Insitute for Information Systems, Univ. of California, San Diego, 1978.

[5] R.L. Sites, personal communication, APIS Dept., Univ. of California, San Diego, 1978.

D20

N79-12735

DESIRABLE FLOATING-POINT ARITHMETIC AND ELEMENTARY FUNCTIONS FOR NUMERICAL COMPUTATION

T.E. Hull, Department of Computer Science
University of Toronto
Toronto, Ontario, Canada

ABSTRACT

The purpose of this talk is to summarize proposed specifications for floating-point arithmetic and elementary functions. The topics considered are: the base of the number system, precision control, number representation, arithmetic operations, other basic operations, elementary functions, and exception handling. The possibility of doing without fixed-point arithmetic is also mentioned.

The specifications are intended to be entirely at the level of a programming language such as Fortran. The emphasis is on convenience and simplicity from the user's point of view. Conforming to such specifications would have obvious beneficial implications for the portability of numerical software, and for proving programs correct, as well as attempting to provide facilities which are most suitable for the user. The specifications are not complete in every detail, but it is intended that they be complete "in spirit" - some further details, especially syntatic details, would have to be provided, but the proposals are otherwise relatively complete.

THERE HAS BEEN A GREAT DEAL OF PROGRESS during recent years in the development of programming languages. However, almost all of this progress has been under the general banner of "structured programming" and almost no attention has been paid to those aspects, such as the semantics of floating-point operations, that are of special interest to practitioners who are interested in numerical computation.

The purpose of this talk is to propose some specifications for floating-point arithmetic and elementary functions. The main design goal is to produce a set of specifications which is most desirable from a user's point of view. There is of course no claim that the set is unique. In fact, many details, especially syntatic details, have been omitted because there are obviously so many possible variations that would be equally acceptable.

It should be emphasized that the specifications are intended to be entirely at the level of a programming language such as Fortran. For example, in discussing arithmetic operations, our concern is entirely with the syntax ard semantics of the programming language expressions.

We feel that it is important to consider such specifications for floating-point arithmetic and elementary functions. Indeed, users who are interested in numerical computation have an obligation to try to reach a consensus on such specifications, unless they are prepared to put up forever with whatever facilities the manufacturers and language designers happen to provide. If some sort of consensus became possible, it could evolve towards a standard. And with the technology changing as rapidly as it is, such a standard may not be too difficult to achieve, or at least to approach much more closely than is the case at present. In any event, with a language standard agreed upon, we would at least have a basis against which we could judge the appropriateness of various trade-offs, and to judge the suitability of any new hardware designs that are being proposed.

The usefulness of a standard in terms of portability of numerical software, and particularly in terms of portability of proofs about what the software does, is obvious.

An ideal arithmetic system should be complete, simple and flexible. Completeness means that the programmer knows what will happen under any circumstance Simplicity leads us to conclude, for example, that the base should be 10. For simplicity we also argue for a particular way of determining the precision at which calculations are performed. We choose a flexible way of controlling precision, and also a flexible mechanism for coping with exceptions such as overflow and underflow.

An ideal system for elementary functions is more difficult to agree upon. Completeness, in the sense of always producing the same results whenever the precisions are the same, would be desirable here too, but probably not practical. However, what is more to the point at this stage is that we emphasize simplicity, and this leads us to require only a single simply-stated accuracy requirement for all elementary functions. In particular, we argue against insisting that a long list of additional special properties be required to hold.

The following is only a summary of what is proposed. Most of the justification for these recommendations has had to be omitted because of space limitations.

BASE

Much has been written about different number bases, and their relative merits with respect to efficiency of storage, roundoff behavior, and so on. We believe that simplicity and convenience for the user should be the primary consideration and this means that

the choice of base is 10. (1)

With this choice, a number of annoying problems disappear immediately. For example, the "constant" 0.1 will really be exactly one tenth. The compiled value for a number will not differ from its input value, and input-output will be simplified in general.

Programmer efficiency will improve if the programmer does not have to keep in mind the peculiarities of other number bases. It may even happen that a base-10 system, including a base-10 internal representation, would turn out to be, overall, the most efficient, besides being the simplest and most convenient for the user.

PRECISION

Earlier versions of what is proposed in this section, including the description of a preprocessor for implementing the main ideas, have been discussed elsewhere by Hull and Hofbauer (2,3)*.

*Nos. in ( ) designate References at end of paper.

It is important that the user have control over the precision. In an ideal system, we believe that

the user should be able to specify separately the number of digits to be used for the exponent of his floating-point values, and the number of digits to be used for the fraction part. (2)

Ideally he should be able to make a declaration such as FLOAT(2,12) X and as a result have the value of X composed of a 2-digit exponent part along with a 12-digit fraction part.

It should also be possible that

variables or expressions, as well as constants, be allowed in the declarations. (3)

For example, FLOAT(2,I+1) X would have the obvious meaning.

The most important part of our proposal with respect to precision is that

the user should be able to specify the precision of the operations to be carried out on the operands, quite apart from, and independently of the precision of the operands themselves. (4)

For example, he should be able to write something like

```
BEGIN PRECISION(3,14)
      ------
      ------
      Y = X + .51 * SIN(X)
      ------
END
```

and mean that every operation in the expression is to be carried out in (3,14)-precision arithmetic, the result of the calculation finally being adjusted to fit the precision of Y, whatever the precision of Y has been declared to be, before the result is assigned to Y.

It is of course intended that

the precision of such "precision blocks" be allowed to change between one execution of a block and the next. (5)

Examples are given in the references by Hull and Hofbauer referred to earlier; however, the preprocessor mentioned there handles only the special case in which only the fraction parts (of the variables and precision blocks) are declared, and their values denote numbers of word lengths rather than numbers of decimal digits.

The specifications we propose for precision control provide a considerable degree of flexibility. In particular, they allow the user to carry out intermediate calculations in higher precision (as may be required, for example, in computing scalar products, or in computing residuals), and they allow for the repetition of a particular calculation in different precisions (as is required, for example, in some iterative procedures, or in attempting to measure the effect of roundoff error).

The proposed specifications are also simple. For example, separating the precision of the operations from the precisions of the quantities entering into the calculations avoids having to remember a lot of rules about how quantities of different precisions combine. (No satisfactory rules for such calculations can be devised in any event, for example, no such rules would enable us to compare the results of doing a particular calculation twice, at two different precisions.)

It must be acknowledged that very high precision calculations would be used only rarely. This means that all precisions up to something like (2,12) or perhaps (3,15) should be done very efficiently, but, beyond that, a substantial reduction in efficiency would be quite acceptable.

One point is worth emphasizing. It is intended that precision 12, say, means exactly precision 12, and not at least precision 12. We cannot measure roundoff error if precision 12 and precision 15 give the same results.

One further point is perhaps worth mentioning. Our requirements for precision control could lead to thinking of the machine as being designed to handle character strings, a number being just a special case in which most of the characters in a string are decimal digits. However, as indicated earlier, we are concerned here primarily with the functional specifications, and not with any details about how those specifications are to be implemented.

REPRESENTATION

Quite independently of how the base is specified, or of what sort of flexibility is allowed with the precision, it is possible to state specific requirements about the representation of floating-point numbers. We will describe what we consider to be desirable requirements in terms which may appear to be hardware specifications but the proposal is not meant to restrict the details of the hardware representation in any way except in so far as the results appear to the user.

The proposal is that

a sign and magnitude representation be used for both the exponent part and the fraction part, and that the fraction part be normalized. (6)

The reason for proposing a sign and magnitude representation is that it is simple, and probably easiest to keep in mind. The reason for allowing only normalized numbers is so that the fundamental rule regarding error bounds that is discussed in the next section can then be relatively simple.

We deliberately do not propose any axioms, such as "if x is in the system then so is -x", to be satisfied by the numbers in the system. Any such statements that are valid are easily derived, and there is no need to state them explicitly. In fact, it might be somewhat misleading to begin with statements of this sort and perhaps give the impression that one might be able to derive the system from a collection of such desirable properties.

Besides the normalized floating-point numbers proposed above

it will be necessary to allow a few other values as well, such as OVERFLOW, UNDERFLOW, ZERODIVIDE, INDETERMINATE, and UNASSIGNED to be used in special circumstances. (7)

We will return to this question in a later section when we discuss the requirements for exception handling.

Although what we have proposed as allowed values for floating-point numbers is, for the purpose of simplicity, very restricted, the hardware can carry much more in the way of extended registers, guard digits, sticky bits, and so on, if that should be convenient for meeting the requirements of the following sections However, if this is done, it will ordinarily be only for temporary purposes, and, in any event, the user would under no circumstances have access to such information. (We are continuing to think of the user as programming in a higher level language such as Fortran.)

ARITHMETIC OPERATIONS

Whatever the base or method of representation, we can still be precise about the kind of arithmetic that is most desirable. For various reasons we propose that,

in the absence of overflow, underflow, indeterminate, and zero-divide, the results of all arithmetic operations be properly rounded to the nearest representable number. (Some further detail is needed to make this requirement completely precise. In case of a tie, we might as well have the normalized fraction part rounded to the nearest even value.) (8)

There are several reasons for preferring this specification:

(a) It is simple, and easy to remember.

(b) Since unnormalized numbers are not allowed, the basic rule required for error analysis is easy to derive and, in the absence of overflow, underflow, indeterminate, and zero-divide, takes the simple form:
$fl(x \circ y) = (x \circ y)(1+\varepsilon)$,
where $\circ$ is an operation and $|\varepsilon| < u$, u being the relative roundoff error bound for the precision that is currently in effect.

(c) Rounding is better than chopping, not because the value of u is smaller (although that happens to be the case), but primarily because of the resulting lack of bias in the errors.

There is a considerable advantage to stating directly what outcome one is to expect from an arithmetic operation, and then deriving any properties that one needs to use, rather than to start off with a list of desirable properties. For example, from the simple specification we have given, it is a straightforward matter to prove that (sign preservation):

$(-x)*y = -(x*y)$,

or that (monotonicity).

$x \le y$ and $z \ge 0$ implies $x*z \le y*z$.

It is misleading to write down a list of such desirable properties and to suggest that rules might be derived from them. (After all, if we did write down all of the most desirable properties we would of course want to include associativity!)

It is undesirable to allow any exceptions to the specifications - even such small ones as the exception to true chopping arithmetic that occurs with IBM 360/370 computers. Nor should we tolerate the inclusion of tricks, such as evaluating A+B*C with at most one rounding error The reason is that it is important for the user to know what happens under all circumstances. A simple rule, that is easy to remember and to which there are no exceptions, is a good way to ensure this knowledge.

To complete the programming language specifications with regard to floating-point arithmetic, we also require that

some conventions be adopted, such as the left to right rule for resolving ambiguities in expressions such as A+B+C. (9)

A further discussion of overflow, underflow, etc., is also required, but that will be postponed to the section on exception handling.

OTHER BASIC OPERATIONS

Besides the arithmetic operations, a programming language must of course also provide various other basic operations. These should include such standard operations as

absolute value
the floor function,
quotient, remainder,
max, min, (10)

as well as

the relational operators. (11)

With the latter it is essential that they work properly over the entire domain, and that, for example, nothing ridiculous happen such as allowing IF(A > B) to cause overflow.

There would also be a need for functions to perform special rounding operations, such as

round the result of an arithmetic operation to a specified number of places in the fraction part,
round up, or round down, similarly,
round a result to a specified number of places after the point (12)

and to carry out other special operations, such as

get precision of fraction part,
get precision of exponent part. (13)

Finally, a special operation may be needed to denote

repeated multiplication. (14)

The purpose of this operation is to distinguish x^n, where n is an integer and it is intended that x be multiplied by itself n-1 times, from the case where it is intended that x^n be approximated by first determining log x and then computing $e^{n \log x}$. Being able to make this distinction would be helpful in calculating expressions such as $(-1)^n$, or $(3.1)^3$. But whether this part of the proposal is accepted depends to some extent on how strongly one feels about dropping the fixed-point or integer type, as mentioned in a later section.

ELEMENTARY FUNCTIONS

For the elementary functions, such as SQRT(X), EXP(X), SIN(X), etc., we propose some simple but uniform requirement such as

$fl(f(x)) = (1+n_1\varepsilon)f(x(1+n_2\varepsilon))$
over appropriate ranges of x, where n_1
and n_2 *are small integers. (Of course,* (15)
each ε *satisfies* $|\varepsilon| < u$, *and the value of u depends on the precision.)*

It would be a nice feature if the n's were relatively easy to remember. For example, it might be possible to require $n_1 = 2$ for each function, and $n_2 = 0$ for at least most of the functions of interest. Unfortunately, the "appropriate ranges" will differ, although they will be obvious for some functions (for example, they should contain all possible non-negative values of x for the square root function).

There is a temptation to require more restrictions on the approximations to the elementary functions, such as

SIN(0) = 0, COS(0) = 1
LOG(1) = 0, ABS(SIN(X)) ≤ 1

or that some relations be satisfied, at least closely, such as

SQRT(X^2) = X,
$(SQRT(X))^2$ = X,
SIN(ARCSIN(X)) = X,
$SIN^2(X) + COS^2(X) = 1$,

or that some monotonicity properties be preserved, such as

$0 \le X \le Y$ implies SQRT(X) $\le$ SQRT(Y)

A few such properties follow from the proposed requirement (for example, SIN(0) = 0), but we propose not requiring anything beyond what can be derived from the original specification. This proposal is made in the interests of simplicity. The original specification is easy to remember, and any proofs about what programs do should depend only on a relatively few "axioms" about floating-point arithmetic and the elementary functions. No one is required to remember a potentially long list (and perhaps changing list!) of special properties of the elementary function routines.

In those cases where something special is required, one possibility is that the programmer take appropriate measures. For example, if it appears that we might want to require that $|\sin(x)| \le 1$, as we might in trying to approximate the integral $\int_0^{\pi} \sqrt{1-\sin x}\, dx$, we can simply replace 1-sin x with $|1-\sin x|$. Alternatively, separate function subroutines can always be developed in order to provide function approximations that satisfy special properties; for example, there could be a special sine subroutine, say SSIN, which produces approximations to sin(x) with special properties such as being guaranteed not to exceed 1 in absolute value.

EXCEPTION HANDLING

Overflow, underflow, indeterminate, and zero-divide have already been mentioned. (It may be that one would like to make further distinctions here, between positive and negative overflow, for example) It should be pointed out that overflow and underflow can occur when precision is changed, especially if the user can change the exponent range. Other exceptions that can arise include trying to compute with an as yet unassigned value, or using a function argument that is out of range.

The first rule should be that,

if an exception arises and the programmer makes no special provision for handling it, the computation should be stopped, along with an appropriate message about where and why. (16)

If the user is aware that an exception might arise, and knows what he wants to do about it, he can often "program around" the difficulty. One example has already been mentioned in connection with an argument getting out of range in $\sqrt{1-\sin x}$. Another arises in trying to calculate min(|y/x|,2) where y/x might overflow. However, such strategies are often quite confusing and sometimes not even available Some kind of general capability for handling exceptions is needed.

Our second rule with regard to exception handling is therefore that

the user should be able to specify the scope over which he is prepared to state what is to be done, and that he be able to detect the cause of the interrupt, in a way such as is suggested in the following. (17)

```
BEGIN
    ON(OVERFLOW)
                    } what to do in case of over-
    (UNDERFLOW)       flow
    (                )
    END
END                 } scope
```

Besides OVERFLOW and UNDERFLOW, the other possible causes of interrupts are ZERODIVIDE, INDETERMINATE, UNASSIGNED, and OUTOFRANGE (i.e., argument of a function out of range).

Third, it is to be understood that

control will be returned to the point of interruption, after the specified action has been taken, unless the programmer has provided for an alternative to be followed, such as stopping the calculations altogether, or perhaps making an exit from that block of instructions. (18)

Fourth, it is also proposed that

the programmer be able to assign a value to RESULT as part of the action to be taken. For example, he could write (19)

```
ON(OVERFLOW) RESULT = 10**50
  (UNDERFLOW) RESULT = 0
END
```

Not allowing the user to have access to the operands, other than through his access to the program variables themselves, has been deliberate. In particular, if the operands that caused the interrupt were "temporaries", it is difficult to see how he could make use of them.

FIXED-POINT ARITHMETIC

In conclusion, we would like to comment that, at least to us, it appears that we do not need to have any type of arithmetic in our programming language other than the floating-point arithmetic described in the preceding sections (except, of course, for complex arithmetic). In particular, there does not appear to be any compelling need for fixed-point or integer arithmetic.

ACKNOWLEDGEMENTS

Much of this work is based on material that was prepared for discussion by members of the IFIP Working Group 2.5 on Numerical Software, beginning with the Oak Brook meeting in 1976 (1). I would like to acknowledge the helpful criticism of members of that committee, particularly of W.S. Brown and T.J. Dekker. I would also like to thank W.M. Kahan especially; although we have not always been in complete agreement, he has certainly had an enormous influence on my thinking about this, as well as many other subjects!

REFERENCES

1. T.E. Hull, "Semantics of Floating Point Arithmetic and Elementary Functions", Portability of Numerical Software (ed. W.R. Cowell), Springer-Verlag, N.Y., 1977, pp.37-48.

2. T.E. Hull and J J Hofbauer, "Language Facilities for Multiple Precision Floating Point Computation, with Examples, and the Description of a Preprocessor", Tech Rep. No.63, Dept. of Comp. Sci., Univ. of Toronto (1974).

3. T.E. Hull and J.J. Hofbauer, "Language Facilities for Numerical Computation", Mathematical Proceedings of the ACM-SIAM Conf. on Mathematical Software II (ed. John R. Rice), Academic Press, N.Y. 1977, Proceedings of the ACM-SIAM Conf. on Mathematical Software II, Purdue Univ. (1974), pp.1-18.

omit

Handling of Floating Point Exceptions

Thomas W. Eggers
Judson S. Leonard
Mary H. Payne

Digital Equipment Corporation
Maynard, Massachusetts

SUMMARY

An IEEE subcommittee on the standardization of microprocessor floating point arithmetic has a proposal under discussion. Part of that proposal concerns overflow and underflow exceptions.

The proposal calls for a "gradual" underflow implemented with denormalized numbers. For a sequence of addition/subtraction operations, the gradual underflow works very well: it almost has the effect of a machine with infinite range numbers. But if an addition/subtraction sequence is interrupted by a multiply or divide, things don't work nearly as well, and a fallback to symbolic information is likely. The proposal helps overflow hardly at all.

The Payne alternate proposal handles overflow, underflow, addition/subtraction, and multiplication/division equally well. It relies on a pointer scheme that is invoked when an overflow or underflow exception occurs. The excepted result is replaced by a reserved operand. The reserved operand encodes information about the exception, stores the "excess exponent," and points to a second word which stores the correct number less the excess exponent factor. Whenever a reserved operand is encountered during execution, a trap occurs and an interpreter performs the operation using the excess exponent for extended range.

The second words that are created on an overflow or underflow exception are held in a table which dynamically changes in size. The table can be managed with traditional storage allocation techniques. It can grow to a maximum size equal to the number of floating point variables in the program. The expected size and access rate of this table are being investigated.

The authors believe that the Payne pointer scheme offers an improvement in both functionality and simplicity over the gradual underflow mechanism.

D21

N79-12736

"PROGRAMMING EFFORT" ANALYSIS OF THE ELLPACK LANGUAGE

John R. Rice
Division of Mathematical Sciences
Purdue University

ELLPACK is a problem statement language and system for elliptic partial differential equations (PDEs) which is implemented by a Fortran preprocessor. ELLPACK's principal purpose is as a tool for the performance evaluation of software. However, we use it here as an example with which to study the "programming effort" required for problem solving. It is obvious that problem statement languages can reduce programming effort tremendously; our goal is to quantify this somewhat. We do this by analyzing the lengths and effort (as measured by Halstead's "software science" technique) of various approaches to solving these problems.

A simple ELLPACK program is shown below to illustrate the nature of the ELLPACK language. Space does not allow a description of the language but it is somewhat self explanatory. See [2] and [3] for further details.

```
*        ELLPACK 77 - EXAMPLE 4 FOR SIGNUM CONFERENCE
EQUATION.      2 DIMENSIONS
   UXX$ +6.UYY$ -4.UY$ +(DUB9(X)-3.)U = EXP(X+Y)*DUB9(X)*(2./(1.+X)-1.)
BOUND.     X = 0.0      , U = TRUE(0.0,Y)
           Y = 1.0      , UY= EXP(1.+X)*SQRT(DUB9(X)/2.)
           X = EXP(1.)  , U = TRUE(2.71828182846,Y)
           Y = 0.0      , MIXED = (1.+X)U  (1.+X)UY  = 2.*EXP(X)
GRID.      UNIFORM X = 5      $      UNIFORM Y = 7
*
DISCRETIZATION(1).   5-POINT STAR
DIS(2).              P3-C1 COLLOCATION
INDEX(1).            NATURAL
INDEX(2).            COLLOCATE BAND
SOL.                 BAND SOLVE
OUTPUT(B).           MAX-ERROR   $   MAX-RESIDUAL
OUTPUT(99).          TABLE(5,5)-U
SEQUENCE.               DIS(1) $ INDEX(1) $ SOLUTION $ OUTPUT(B)
                        DIS(2) $ INDEX(2) $ SOLUTION $ OUTPUT(B)
                        OUTPUT(99)
OPTIONS.             MEMORY $ LEVEL=2
FORTRAN.
      FUNCTION TRUE(X,Y)
      TRUE = EXP(X+Y)/(1.0+X)
      RETURN
      END
      FUNCTION DUB9(T)
      DUB9 = 2./(1.+T)**2
      RETURN
      END
END.
```


C-2

A problem solution with the ELLPACK system goes through three principal stages: (1) the ELLPACK language input is read by a Fortran preprocessor which writes a Fortran Control Program, (2) the Control Program is compiled, and (3) the Control Program object deck is loaded along with modules from the ELLPACK library which implement steps in the solution of the PDE. We compare the programming effort for each of these steps, i.e., (1) an ELLPACK statement of the PDE problem to be solved and method to be used, (2) preparation of the Control Program, assuming familiarity with the module library and (3) programming the entire solution in Fortran.

Three measures of programming effort are used: lines of code, total number of operators and operands and "effort" measured by thousands of elementary mental discriminations. The latter two measures are part of Halstead's "software science" presented in [1]. This is an empirical method to define and relate various program parameters to the effort required to write the programs. While we do not attempt to explain this method, it is very plausible that the total number of operators and operands in a program is more directly related to the complexity of a program than the number of lines of Fortran. Two short-comings of the method for this application are (1) that it ignores declarations and I/O statements and (2) the mechanism for estimating the effort for a set of tightly integrated subroutines is inadequate. However, the measurements are good enough for the present purposes where only rough accuracy is needed.

We consider 10 example problems and present the data N=total number of operators and operands, L=total lines of code (including comments in the Fortran modules, most of which are well commented), C=code complexity measured by number of operators and operands per line, and E=programming effort in 1000's of elementary mental discriminations as defined by Halstead. For each problem we have data for the ELLPACK language (labeled ELPK), the Control Program (labeled Control) and the set of library subroutines used (labeled Modules).

	PROBLEM 1			PROBLEM 2			PROBLEM 3			PROBLEM 4		
	ELPK	Control	Modules	ELPK	Control	Modules	ELPK	Control	Modules	ELPK	Control	Modules
N	187	1793	14,349	103	1331	6632	147	1552	14,203	134	1354	12,671
L	33	381	3,852	22	295	1330	27	353	5,348	29	314	3,402
C	5.7	4.7	3.7	4.7	4.5	5.0	5.4	4.4	2.7	4.6	4.3	3.7
E	27	1076	6,425	5	371	4804	14	852	4,232	12	614	5,881

	PROBLEM 5			PROBLEM 6			PROBLEM 7			PROBLEM 8		
	ELPK	Control	Modules	ELPK	Control	Modules	ELPK	Control	Modules	ELPK	Control	Modules
N	113	1231	11,198	125	1358	15,113	125	1366	8500	102	1238	7,261
L	40	303	2,918	42	336	5,435	51	311	2561	29	303	2,145
C	2.8	4.1	3.8	3.0	4.0	2.6	2.5	4.4	3.3	3.5	4.1	3.4
E	8	385	5,306	12	587	3,784	11	444	2771	6	394	2,211

	PROBLEM 9			PROBLEM 10		
	ELPK	Control	Modules	ELPK	Control	Modules
N	112	1283	14,134	87	1716	7997
L	38	315	3,937	110	365	2517
C	2.9	4.1	3.6	.8	4.7	3.2
E	6	503	6,739	4	390	3243

There are considerable variations among these examples but there is also an obvious trend of greatly increased "length" from stage to stage, no matter how it is measured. The programming effort E should increase faster than the number of lines, but it does not always do so because of the inability of the software science method to completely account for the use of modularity in implementing an algorithm.

Comparing the Control and Modules data should be representative of the comparison of using or not using a library of powerful subroutines. We see that the ratios of effort range from 6 to 15 with 10 as an average, the ratios of lines range from 6 to 17 with 11 as an average. Thus we conclude that, at least in the context of solving PDEs, the use of a library increases programming productivity by a factor of 10. It may well increase it more and the quality of the results will be improved if the library is good.

Comparing the ELPK and Control data should measure the value of a problem statement language compared to using a library. The ratios of effort range from 40 to 100 with 60 as an average and the ratios of lines range from 3 to 13 with 9 as an average. We thus conclude that using an ELLPACK type preprocessor increases programming productivity by a factor of 10 to 50.

We also conclude that using this preprocessor instead of writing the programs from scratch reduces programming effort by a factor of between 100 and 500.

This work is partially supported by NSF Grant MCS76-10225.

[1] M. H. Halstead, Elements of Software Science, Elsevier North-Holland, New York, 1977.

[2] J. R. Rice, ELLPACK: A Research Tool for Elliptic Partial Differential Equations Software in Mathematical Software III (J. R. Rice, ed.) Academic Press, 1977, pp. 319-341.

[3] J. R. Rice, ELLPACK 77 User's Guide, CSD-TR 226, Computer Science Dept., Purdue University, September 1978.

OMIT

Notes from the Second Department of Energy Library Workshop

by

Kirby W. Fong - National Magnetic Fusion Energy Computer Center at the Lawrence Livermore Laboratory

and

Rondall E. Jones - Sandia Laboratories Albuquerque

Part I – A General Review of the Workshop

The U S Atomic Energy Commission (AEC) and its successors, first the U.S Energy Research and Development Administration and now the U S Department of Energy (DOE) and Nuclear Regulatory Commission, has been and continues to be one of the nation's major purchasers and users of large scale scientific computers Historically, each of the more than dozen computer centers at different laboratories evolved independently of the others so that each was self-contained In particular, each computer center developed mathematical software libraries according to its own needs In 1975, representatives for the mathematical software libraries at the various AEC computer centers met, with Argonne National Laboratory as the host, to hold the first Workshop on the Operational Aspects of Mathematical Software Libraries. Among the purposes of the first Workshop were: (1) to meet colleagues doing similar work at other AEC computer centers, (2) to share experiences in the management of mathematical software libraries, and (3) to discuss ideas and issues in the operation of libraries The first Workshop was sufficiently rewarding that the participants felt it appropriate to hold a second Workshop in three years, an interval that would encompass sufficient progress in library operation that new experiences and ideas could be discussed.

The Second DOE Workshop on the Operational Aspects of Mathematical Software Libraries was hosted by the National Magnetic Fusion Energy Computer Center at the Lawrence Livermore Laboratory in August 1978 It was attended by thirty-five participants representing fifteen DOE computer centers. Representatives from three non-DOE computer centers were also invited A major new development in DOE computer centers, the use of commercial mathematical software libraries, led to inviting representatives from three major commercial library companies.

The Workshop included both individual presentations and group discussions. We will deal here with only the group discussions because they reflect the problems and issues that all of us, in or out of DOE, face in the management of mathematical software libraries.

One problem regarded with varying degrees of concern by the participants is the proliferation of mini computers While some mini computers are limited by small amounts of memory or inaccurate elementary functions, these appear not to be the principal problems The problem is that there are potentially so many brands and models at any given site, each being used for scientific computation. Consequently, the mathematical software librarian has the task of supplying many versions of his or her library – one for each mini computer.

At the opposite end of the spectrum is the super computer with unconventional architecture At this time, the only DOE computer centers acquiring such machines already have computers with long word lengths. This means extensive conversion between single and double precision has not yet been needed The basic problem is that standard Fortran codes on the existing large computers may be unable to take full advantage of the new architecture of the super computers This problem is currently felt to be handled best by increased modularization (*e g* the Basic Linear Algebra Subroutines or BLAS) so that only a limited number of key modules need to be rewritten (presumably in assembly language) for a new super computer. Conspicuous by its absence was any mention of programming languages other than Fortran Apparently, no DOE computer center expects to use any language other than Fortran or a limited amount of assembly language in mathematical software libraries be they libraries for mini computers, conventional large computers, or super computers

Participants were asked to mention mathematical areas in which current libraries seemed to lack sufficient coverage. In many cases, the areas mentioned below reflect a single user at a single site asking for a capability rather than a wide spread need. (1) Sparse linear algebra routines of all types are in demand This was not an isolated occurrence. (2) Certain areas of physics require high accuracy quadrature routines in up to twelve dimensions. (3) One computer center reported a need for Fortran callable subroutines to perform symbolic algebra

and produce Fortran compilable output. (4) There is a modest need for multi-dimensional surface fitting and approximation routines where data are given at co-ordinates that do not constitute a rectangular mesh – *i.e.* randomly distributed data points. (5) There is no end to the special functions that users request Users are asking for functions of two and three parameters with fractional or even complex degree, order, or argument. (6) Users are starting to encounter more multi-point boundary value problems We anticipate greater demand for routines in this area.

The statistics software area is one in which the responsibilities of the mathematical software librarian are not well defined. It appears that the practitioners of statistical analysis prefer to use self-contained tools – *i.e.* programs that handle input, output, and graphics as well as analysis Many such complete programs exist and are commercially available. Each is designed to handle some reasonable subset of problems. A library of statistics subroutines is therefore needed only when a statistician must construct a new tool for a new class of problem, and then he or she will also need input, output, and graphics routines as well as statistics routines. We believe it fair to say the discussion of this topic was inconclusive.

The era in which each computer center could afford to write and maintain its own complete mathematical software library is now generally acknowledged to be past. The continuing expansion in numerical analysis precludes having experts in every area of numerical analysis on the staff, and DOE is also sensitive to the duplication of effort implied by having independent library efforts at many computer centers. Thus commercial libraries are seen as playing a larger role in DOE computer centers. They can provide and support high quality, standard, general purpose numerical software while allowing the staff at each computer center to specialize in mathematical software unique to the needs of users at each computer center. The second Workshop therefore invited three of the major commercial library companies (IMSL, NAG, and PORT) to send representatives. The representatives were asked to give informal descriptions of their current activities and plans

From the viewpoint of DOE participants, probably the most important benefit from having the commercial representatives was the chance to tell them how DOE perceived their roles and to state specifically what services the DOE computer centers would be needing. The commercial libraries play a role somewhat analogous to the scholarly journal They are increasingly being viewed as the vehicle through which the author of mathematical subroutines presents his or her software to the world just as a journal is used for presenting research to the world We expect that software will be refereed just as journals articles are refereed in order to achieve a uniformly high quality of content. Specific needs of computer centers focus primarily on documentation. As the completely home grown library recedes into history, so will the completely home grown library document. It will be necessary for commercial libraries to supply documentation that is convenient to use and can fit into the great variety of procedures which different computer centers use to maintain documentation Most DOE computer centers are still relying on hardcopy (*e.g.* paper) documents to some extent, but the cost or inconvenience of printing new manuals or inserting revision pages is pushing computer centers in the direction of machine readable, on-line documentation. Typing a document into a form suitable for processing by a report editor is not vastly more expensive than typing a camera ready master, and it means the document is stored in a form that permits revisions to be made easily and quickly If the document is kept on-line, the user can interrogate it interactively or print a copy of the current write-up on some output device The on-line document is not free of problems however One is that on-line storage (*e.g.* disks) is expensive. We await a decrease in hardware costs combined with the inevitable increase in labor costs and inconvenience in maintaining paper documents to tilt the balance in favor of on-line documents Computer networks with users who are geographically dispersed have already had to shift away from a dependence on hardcopy manuals A second reason is that limited character sets in various output devices (*e.g.* printers or terminals) prevent writing mathematical expressions in the natural way. The commercial library company faces exactly the same problems as its customers in producing and maintaining documentation Just as business offices have resorted to word processing equipment to control the cost of producing correspondence, so the commercial libraries will have to turn to the computer for assistance in producing and maintaining documentation. Their problem is more difficult than that of any individual computer center in that they must furnish portable, machine readable documentation that can be integrated easily into each customer's system and set of output devices. Currently, they furnish a finished, hardcopy manual The problem is not the content of the manual; it is the format. The current form makes it impractical for computer centers to furnish a copy to every user Also, few users really need to own a complete manual. Rather they need to be able to extract and print a subset of the manual appropriate for their needs. It is now generally recognized that documentation is an integral part of mathematical software (software without documentation is nearly useless) but that construction, maintenance, and distribution of the library document is perhaps a more intractable problem than construction, maintenance, and distribution of the library itself. With commercial libraries now containing hundreds of subroutines come manuals containing thousands of pages. The sheer bulk of documentation, while it suggests automation is now in order, also means the companies must choose a documenation system carefully because they can afford to type such large documents only once. Thereafter, the companies must be able to transform their machine readable master documents into various "documentation system ranges" as needed by their customers.

Numerous other topics received varying amounts of attention at the Workshop (1) Participants were asked to mention any mathematical software packages which might supplement general purpose libraries. (2) Participants were asked to describe any tools or

preprocessors that they had found useful in preparing, testing, and documenting mathematical software. (3) One of the Workshop participants is also a member of the DOE Advanced Computing Committee Language Working Group He spoke about the activities and plans of the group and answered questions about why DOE was so interested in designing a Fortran which would be an extension of ANSI 1977 Fortran. (5) Several DOE computer centers related their experiences and observations in monitoring the use of mathematical libraries (6) The concept of a core library was advanced as a response to some of the constraints now being encountered A core library is of manageably small size and includes routines from any source, local or outside, to provide coverage of most of the mathematical algorithms needed by users A core library is small enough to be supported by a small staff and does not require a massive manual The staff is free to adopt for the core library the finest routines from commercial libraries, outside authors, or, local authors (7) Participants reviewed briefly the means for exchanging information or software. The ARPA net is not available to all computer centers, but it does allow transmission of messages and files between some of the DOE computer centers The magnetic tape formats recommended by the Argonne Code Center were agreeable to most participants although some thought that seven track tapes should be blocked instead of unblocked (8) Finally, the Workshop gave librarians a chance to express their advice for authors who would write mathematical software for libraries. This is in contrast to mathematical software packages intended for stand alone use

Part II - Suggestions for Authors of Mathematical Software

In this part of the paper we present our views about the format for Fortran software to be admitted to libraries. Most of these ideas were presented at the DOE Library Workshop; however, we do not wish to imply that they were endorsed by the Workshop Most of these ideas come from the discussions between Sandia Laboratories, the Los Alamos Scientific Laboratory, and the Air Force Weapons Laboratory about the feasibility of constructing a common mathematical library from existing software These discussions were organized under SLATEC, a committee with representatives from the three laboratories, which co-ordinates technical exchanges among the three members.

The programming environment for the development of mathematical software is influenced to some extent by what the author perceives as the manner in which the software will ultimately be used. In particular, a mathematical package that is destined for stand alone use will be written to be self-contained, that is, it will use no externals other than the basic (Fortran) externals and will try to avoid any input/output or system dependence Such attention to portability is commendable, for it eases the recipient's task of installing the package; yet, a librarian may nevertheless be unhappy because the package adds yet another linear system solver or error message printing routine to the library From the librarian's point of view, a collection of completely independent routines is not a library. A collection of software cannot really be elevated to the status of library until redundancy is minimized, error conditions are handled in a systematic way, and the routines and their documents are presented in a fairly uniform way Let us be more specific about the attributes the librarian values

(1) Whenever possible, the arguments for a subroutine should be in this order: (1) input, (2) input/output, (3) output, (4) work arrays An exception is that array dimensions should immediately follow the array name. Work arrays should be limited to no more than one of each needed type, *i.e.* one single precision array, one double precision array, one complex array, one integer array, and one logical array. This implies user callable routines may actually be nothing more than interface routines which carve the work arrays into smaller pieces for use by other subroutines.

(2) Each subprogram should have a block of information called a prologue. The prologue should immediately follow the subprogram declaration. The first part of the prologue should be an abstract in the form of comment cards which describes the purpose of the subprogram and gives the author, history, or references for the subprogram The second part should be a description of each argument in the calling sequence, and each argument should be described in the order in which it appears in the calling sequence It has been found that users will code a call by reading the argument description, thus, such a description should not mislead the user into writing actual arguments in an incorrect order The third part of the prologue should be array declarations These may be actual declarations or comment cards describing arrays and their sizes The fourth part of the prologue should be a comment card

```
C *** END OF PROLOGUE
```

which signals the end of information for the user. This type of sentinel is of great use in the automatic preparation of user manuals A string processor or other text editing program can take the cards from each subprogram up to the sentinel to construct a library manual. The prologue should contain sufficient information that it could be used as the primary user document

(3) If at all possible, any separate documentation should be supplied in machine readable form Hardcopy documentation may be suitable when a complete package is supplied to a single user, but library routines are available to all users. Hence a document for a library routine should be in a form where it can be edited easily to fit in a manual or in a form where users can easily print as many copies as they need. Since

many output devices are not capable of printing integral or summation signs and other mathematical notation, considerable ingenuity may be required to write the document using only the ASCII or EBCDIC character set. We furthermore recommend that authors restrict themselvs to the intersection of the ASCII and EBCDIC character sets At this time we are inclined to accept mixed upper and lower case documentation, however, authors who are dedicated to distributing their software may wish to confine themselves to the forty-seven characters in standard Fortran. Line lengths for machine readable documents should not exceed 80 columns as most terminals will print only 80 columns. If the document describes the arguments of a subprogram, it, like the prologue, should describe them in the same order in which they occur in the calling sequence The names of any arguments or internal variables described in the document should be exactly the same as the names in the Fortran subprogram. This assists the user who is symbolically debugging his program who may ask by name for the values of arguments or key internal variables inside the library routine. For example, if EPSLON is an argument, it should be referred to as EPSLON, not EPSILON, in any separate document

(4) A comment card of the following type

C *** FIRST EXECUTABLE STATEMENT

should be placed between the last declaration or arithemtic statement function and the first executable statement This assists string processing programs in inserting CALL statements to system routines that monitor the use of library routines

(5) Input/output (I/O) should be localized if it is present at all. READ and WRITE statements should be confined to one subprogram and not be scattered throughout a package This makes modification of any I/O much simpler. We do not consider the penalty of going through another level of subroutine call to perform I/O a serious penalty.

(6) We recommend that authors use the PORT Library or SLATEC error message packages rather than write their own routines for printing error messages (A description of the latter may be found in Sandia report SAND 78-1189) Both packages are portable. The packages are (by design) very similar in approach, with the SLATEC package differing from the PORT package in some features to reflect the production computing environment in which the SLATEC library is to be used. We suggest that authors choose the package which better suits their philosophy Use of these packages then relieves the author of the burden of designing and coding his or her own error handling procedures

(7) There is some controversy whether machine constants should be computed internally or data loaded in library routines. We prefer that authors use similar if not identical routines to those in the PORT Library. These routines return machine constants which are data loaded inside these routines. This minimizes the number of data statements that must be changed in moving a library from one machine to another. It also precludes the possibility that some new optimizing compiler or architecture might invalidate a tricky algorithm for computing a machine constant

(8) We encourage authors to use reputable, existing software as building blocks for their packages Examples are EISPACK, FUNPACK, LINPACK, IMSL, NAG, and PORT. We also encourage the use of the Basic Linear Algebra Subroutines (BLAS) because they are a small enough set of routines that we can reasonably expect to provide an optimal set for each machine This in turn means that higher level routines calling the BLAS can be made more efficient just by improving the BLAS We thus minimize conflicts between portability and efficiency by isolating efficiency dependent parts of a program into small modules which can be recoded easily

(9) Until some portable subset of ANSI 1977 Fortran becomes recognized, mathematical software should be written in a portable subset of ANSI 1966 Fortran as defined by the PFORT Verifier Authors of routines that do not pass the PFORT Verifier should offer good reasons why their routines should not be modified to do so

(10) Avoid using common blocks because users may accidentally invent program or block names that conflict If common blocks or subroutines internal to a package (not called by users) are used, pick highly unusual names in order to minimize the chance that they will conflict with names existing elsewhere. User callable routines should also have very distinctive names, possibly names that are obviously related to the package, that are not likely to cause conflicts. Examples of bad choices are START, TIME, F, OPEN, CLOSE, FFT, INTEG, SOLVE, SORT, SECOND, INIT, and QUIT. These all have a high probability of conflicting with user or system library names Authors should also avoid names used in widely available software such as IMSL, NAG, PORT, EISPACK, FUNPACK, LINPACK, BLAS, and DISSPLA.

We believe the DOE computer centers are not alone in moving from libraries consisting solely of locally written software to libraries including externally written software We urge software authors who are proud of their products and wish to see them widely used, to consider putting their software in a form that may more easily be integrated into libraries. Not only are computer centers becoming more receptive to outside software in their libraries, they tend to promote their library software more vigorously than software which exists separately on some tape somewhere in the machine room. The "official" library, for example, is usually readily available to the linkage editor in an on-line disk file. Librarians quite naturally will prefer to accept software that fits into libraries easily and has documentation that can easily be transformed into the local canonical form

Activities of the DOE Advanced Computing Committee Language Working Group

Rondall E. Jones, Sandia Laboratories Albuquerque

ABSTRACT

The Language Working Group is a technical arm of the DOE Advanced Computing Committee. The purpose of the Group is to work toward providing a compatible Fortran environment at the ACC sites. A brief history of the efforts of the Group is given, and the general features of the language the group will recommend are discussed. This language is a multi-level Fortran with Fortran 77 as the core.

HISTORY

The Advanced Computing Committee (ACC) is a committee of representatives from the management of the scientific computer rescources at the large DOE research and development laboratories. The function of this committee is to help guide various aspects of the use of large scientific computers at the laboratories represented in the ACC. The ACC Language Working Group (ACCLWG, or LWG) is a subcommittee reporting to the ACC, made up of one or two technical personnel from each site, plus representatives from two closely related non-DOE sites. Approximately twelve persons are currently serving on the LWG. The LWG was formed by the ACC in October 1976, as a technical arm to advise ACC in matters concerning programming languages.

Specifically, the major assignment of the LWG is to advise on how to provide a "compatible Fortran environment" at all the ACC sites. This requirement was motivated by the current situation in which many large programs written at one site cannot be used at other ACC sites because of the considerable differences in the versions of Fortran in use at the various sites. Indeed, it is sometimes not possible to run the same or similar program on two different computers at the same site. It should be pointed out that these differences did not come about casually, but rather were the result of each laboratory's attempt, over the years, to deal most effectively with the advanced computer hardware of which they were often the first recipients.

During the first year of the LWG's existence, the important capabilities of the Fortran languages in use at the ACC laboratories were distilled, and the concept of a multi-level Fortran language based on the new ANSI standard, Fortran 77, was refined. In this multi-level Fortran, the core, or "Level 0", of the language would be precisely Fortran 77 (i.e., X3.9-1978). Level 1 would consist of Level 0 plus features which were "de facto standard" at all the laboratories, or which were clearly desirable by all the laboratories and did not involve any technical questions of implementation. Level 2 would consist of Level 1 plus all the other functional capabilities deemed necessary by the LWG (as determined from the survey of features in use) which were technically feasible for a language usable on a broad class of scientific computers. Level 3 would consist of Level 2 plus necessary features which for some reason could not be placed in Level 1 or 2. Thus, Level 3 features would probably not be applicable to some computers. This concept of a completely nested series of three levels of Fortran, with the current ANSI standard as the core, was presented to the ACC by the LWG at its fifth meeting in November, 1977.

LEVEL 1 FORTRAN

Once the multi-level approach to achieving a compatible Fortran environment was approved, the first business of the LWG was to agree on a detailed description of the Level 1 features. Such "detailed descriptions" do not include a choice of exact syntax to be used, but rather discuss the functional capability itself. Such descriptions were usually made difficult by the fact that the "de facto

standard" features in current use were based on features in the previous ANSI Fortran standard. It was often necessary to revamp these features significantly to base them on Fortran 77. Briefly, the features defined to be in Level 1 are as follows.

1. An asynchronous input/output feature, similar in capability to the well known BUFFER IN/BUFFER OUT feature, but built on Fortran 77's expanded READ/WRITE/INQUIRE features.

2. NAMELIST input/output, in much the same form currently in wide use.

3. Timing functions, including the capability to determine elapsed CPU time as well as time of day, date, and remaining job time left before time limit.

4. Input stream compiler directives to control listing, list suppression, and page ejecting.

5. "Bit-by-bit data manipulation" features including octal and hexadecimal constants and format descriptors, word oriented shifting and masking operations, and bit-by-bit Boolean operations.

The functional description of Level 1 was completed by the LWG's seventh meeting, and was presented to the ACC by the officers of the LWG in May 1978.

LEVEL 2 FORTRAN

The next order of business of the LWG was to develop detailed functional descriptions of the features to be in Level 2. This was a harder task than for Level 1 because the features were in less common use than Level 1 features, and were more technically difficult to fully describe. In addition, it was desired to add fairly detailed examples using illustrative syntax to demonstrate the feasibility of the feature. (Note we do not mean examples of the syntax chosen for a feature, but an example syntax which might or might not eventually be selected.)

At the time of this writing, it appears that all, or almost all, of the features chosen for Level 2 will indeed be written up in detail by the committee's tenth meeting in October 1978, which is the goal which has been set. Briefly, the main features likely to be in Level 2 are as follows. (A more definitive listing should be available by the time of the presentation of this paper.)

1. Array processing, including referencing of whole arrays in assignment statements without the use of subscripts, referencing sections of arrays, array valued conditional assignment statements, and both elemental and transformational array valued functions.

2. Dynamic array allocation, including dynamic array renaming and sectioning, a COMMON-like feature for dynamic arrays, and appropriate environmental inquiry features.

3. A macro capability, in which a macro can be invoked as an expression, a statement, or outside a subprogram. The simplest form of a statement macro would be equivalent to an INCLUDE feature. Macro libraries are allowed, and facilities are included to allow generation of unique statement labels and variables within the body of the macro.

4. More general, "structured" control structures for looping and case selection.

5. Various provisions for improving program form, possibly including such items as a larger character set, trailing comments on a line, longer variable names, multiple assignment statements, and optional automatic indentation.

6. COMPLEX DOUBLE PRECISION type declaration, with appropriate extensions to the intrinsic function set.

7. An environment inquiry feature probably implemented through inclusion of a substantial family of intrinsic functions which provide information on the details of the machine's arithmetic and related matters.

8. A data structuring capability, centered on the concept of a "record" which consists of fields and subfields which are accessible as an aggregate or individually. Among other things, this feature allows very easy acess to part of a word.

9. Expansion of certain features in Level 1, such as extended NAMELIST features.

LEVEL 3 FORTRAN

Clearly, the Level 2 language will be a considerable extension beyond Level 1. Level 3, on the other hand, will probably contain no features beyond Level 2 initially. Rather, a careful definition will be given as to what kinds of features would be included in Level 3 if such a need arises at a later date.

SUMMARY

In summary, the concept of a multi-level Fortran language, with Fortran 77 as "Level 0" was developed in response to the need for a compatible Fortran environment for the ACC sites. A fairly detailed description of the recommended language, including illustrative syntax, but not including final syntax choice, was developed in only about a year from the time the decision was made to go ahead with that development. More detailed language specification will hopefully be performed by a much smaller committee working more intensively for several months. The result of that effort will then be examined by the LWG. It should be emphasized that this effort is oriented to eventually greatly improving the computing environment for the ACC laboratories; it is not an attempt to usurp any function of the ANSI Fortran committee, though the LWG certainly communicates with that committee. Indeed, the philosophy of the LWG would be to restructure its multi-level Fortran to incorporate any future standard as the core of the language.