SLAC-PUB-5583
CERN-DD/89/18
June 1991
(M)

EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH

The Selection and Comparison of Programming Languages for High Energy Physics Applications *

Bebo White

Data Handling Division, CERN
and
SLAC Computing Services

Presented at the Conference on Computing in High-Energy Physics
Oxford, April 1989

Abstract

In this paper I discuss the issues surrounding the comparison and selection of a programming language to be used in high energy physics software applications. The evaluation method used was specifically devised to address the issues of particular importance to HEP applications, not just the technical features of the languages considered. The method assumes a knowledge of the requirements of current HEP applications, the data-processing environments expected to support these applications and relevant non-technical issues. The candidate languages evaluated were Ada, C, FORTRAN 77, FORTRAN 8x, Pascal and PL/I.

Particular emphasis was placed upon the past, present and anticipated future role of FORTRAN in HEP software applications. Upon examination of the technical and practical issues, I have drawn some conclusions and made some recommendations regarding the role of FORTRAN and other programming languages in the current and future development of HEP software.

---

*Work supported by the Department of Energy, contract DE-AC03-76SF00515

1. Introduction

> "For all its inelegance, and lack of safety features, it seems certain that FORTRAN will remain the main language for HEP code well into the 1990s...."
>
> *Computing at CERN in the 1990s* [8]

> "FORTRAN is probably the only perennial standard which will never be questioned."
>
> *Trends in Computing for HEP* [32]

For largely historical reasons FORTRAN has established itself as the "lingua franca" of high energy physics. FORTRAN and HEP have "grown up together" over the past 25 or more years with the result being a large body of excellent and efficient physics analysis and support software. In the development of this software the faults of FORTRAN have been overlooked or cleverly compensated for by the creative people involved. This software represents a considerable investment in time and expertise and, for the most part, *it works!* Future HEP applications can and should take advantage of this historical foundation.

However, a new generation of HEP experiments with more complex data processing requirements, significant advancements in hardware and software technology and more computer-knowledgeable physicists have taxed FORTRAN 77's ability to keep abreast. Newer computer programming languages offer needed functionality not present in FORTRAN 77. Efforts at FORTRAN 8x standardization have not proceeded at the expected pace thereby threatening the time frame within which a version with more powerful and needed features will become available. It has become clear to many that programming language alternatives to FORTRAN should be seriously evaluated without losing the investments of the past.

Physicists have long been recognized as perhaps the most knowledgeable of natural scientists with respect to the technical aspects of computing. Advancements in high energy physics have been driven by advancements in computer hardware and software technology and vice versa. There is significant cross-over of physicists into computer-related tasks. Physicists and developers of physics software have the expertise and should be given a choice of development programming languages as long as there is minimal impact on existing systems and if it can be demonstrated that such a choice would retain or improve software quality.

This paper describes an exercise in which a number of computer programming languages were evaluated specifically for HEP applications. The method devised for this evaluation is based upon a knowledge of the requirements of the applications involved, the data-processing environments expected to support those applications and additional surrounding technical and non-technical issues. This method, coincidentally, closely parallels the feasibility and requirements analysis common to many software engineering methodologies.

2. A Method of Programming Language Comparison

The programming language to be used for any software application is a critical determinant of the speed of software development, the ease of software maintenance and the portability of software to other systems. Many language comparisons have appeared in the literature [10],[11]. The majority of these comparisons have been conducted on the languages *in situ*. Little, it seems, has been written about how languages should be evaluated and assessed with respect to specific software projects.

The elements defined for inclusion in this evaluation of programming languages are:

- Technical specification of the programming application
- Analysis of candidate programming language features
- Programming language fit in the application environment
- Growth and future development

2:1 Technical Specification of the Programming Application

The first step in a programming language evaluation should be conducted before the languages to be evaluated are specifically identified (or at least certainly before any specific language features are considered). The identification of the domains served by particular programming languages (i.e., candidate language selection) depends upon a precise specification of the system for which the software is being developed.

It is important that language features be ignored at this step lest they unduly influence the specification. Once these specifications have been defined, a selection of programming languages which could conceivably be used to implement these specifications are identified. In this way, the programming languages can be examined within an established framework of structured design and programming methodologies with an eye towards features that add or detract from the ability to produce code embodying the goals of these methodologies and the project specifications.

2.2 Analysis of Candidate Programming Language Features

Once candidate languages have been identified, the *features* of those languages with respect to the system specifications are examined. In such an exercise it is virtually impossible to avoid language bigotry. It is clear that no programming language does all things for all people. In all cases it is necessary to make certain compromises. It is most important to remember the fact that all successful programming languages have been designed to describe computations in well understood problem domains and that it is probably unwise to expect any particular programming language to provide a fertile medium for describing computations in more than a few of these domains.

In order to identify accurately specific features of a programming language and to understand the scope of said features, it is *necessary* to have:

- A defining document – the reference manual for the language is essential. Other documents, such as a user guide, may prove helpful by expanding upon and clarifying statements in the reference manual.

- A formal definition – while a formal definition is potentially the ultimate defining document, the underlying assumptions of the definition must be noted. For example, the language defined by the formal definition may be only a subset of the language defined in the reference manual.

It may also be *useful* to have:

- A reference compiler – when the language definition is not complete, the syntax and semantics of a feature may be discovered by using a reference or "standard" compiler for the language.

Using the application specifications and particular language reference materials it is possible to build a "wish list" of those language features of importance. The application specifications help to identify those subsets of specific language features which are relevant to compare. In this manner, the candidate languages are not compared "in toto," the common practice in language comparison which often leads to prejudicial results.

2.3 Programming Language Fit in the Application Environment

This next stage of the evaluation addresses the fact that selection of a language may easily be based upon the ideal, implementation-independent features of that language. However, the hard reality comes in determining the impact of the selection of a particular language and finding a suitable implementation (machine-dependent) of the language. This language *functionality* can be defined in two specific areas of importance:

- the technical issues surrounding the language selection

- the practical issues surrounding the language selection

2.3.1 Technical Issues

In an ideal functional comparison, applications are developed in each language to enable a functional evaluation. Depending upon the motivation for the evaluation and the availability of programmers proficient in the languages, programs may be chosen to illustrate differences in the approaches of the languages or be indicative of the kinds of tasks the language will be used for. Sample programs and test suites can also provide a source of technical data.

Sample applications and functional tests can provide a variety of data. However, such data must be viewed in perspective in that it represents *vendor-specific implementations of a language on specific hardware.*

The function of these tests should be to determine language *usage* in execution environments. A *compiler test* could be done simultaneously, but it is important to differentiate between compiler features (e.g., compilation speed, linking speed, diagnostics, hardware requirements) and language features which have been uniquely identified.

The results of such a technical analysis must be weighted with respect to the role of the implementation-specific environment in which they were derived and the distributed environment of the real-life application. For example, program performance on a supercomputer and a workstation should not be measured in terms of speed of execution, but rather in language functionality on both systems and the similarity in results.

Implementation-specific language extensions should not be ignored in language evaluation. Technical testing allows the evaluation of these extensions and the ramifications their use would present in a distributed system. Technical evaluation should also include the identification and evaluation of applicable language-specific tools and techniques. Many such tools have been developed to compensate for language deficiencies. Tools may come from a variety of sources, usually *not* the compiler vendor. Applicable techniques result from an in-depth knowledge of the language. Descriptions of techniques are widely available in the literature.

2.3.2 Practical Issues

In the area of practical issues, it is the goal to identify the *impact* of the candidate languages if introduced into the *present* application environment. Practical considerations and issues can easily render the technical issues moot.

Identification of practical issues is probably best done in stages with the more obvious issues (cost, staff impact, possible training, etc.) evaluated later. These issues are more difficult to quantify and are usually based on factors which cannot be derived from the language features and technical issues analysis.

3. Programming Language Comparison for HEP Applications

3.1 Specification of a HEP Programming Application

High energy physics data reduction and analysis programs provide good examples of highly numerical-intensive, batch-oriented scientific programming applications. The following operations are typical in these programs and must be supported (or emulated) in programming languages considered for these applications:

- input/output of binary files
- 64-bit floating-point arithmetic
- operations using complex data types
- vector and matrix arithmetics

- data structure manipulation
- access of common blocks of data
- direct addressing of dimensioned variables (as distinct from matrix operations and including non-zero lower bounds for arrays)
- separate or independent compilation of subprograms
- subprogram parameter passing by value and reference and the ability to pass routines as parameters
- access to libraries of mathematical functions
- access to histogramming services
- access to graphics services
- extended file or database services (e.g., particle tables).

Williams [29],[30] lists additional requirements which deserve attention for language evaluation applicable to embedded systems. His list includes concurrent computation and the ability to perform bit manipulation in special purpose hardware. These requirements are incorporated into the previous list of specifications.

3.2 Analysis of Candidate Programming Language Features

The first column of the table in Appendix A is a "wish list" of language features felt to be necessary to satisfy the requirements of a HEP application. It is against this list that the features of the candidate languages are evaluated.

Russell [28] divides language features into two categories, small- and large-scale. Small-scale features have to do with the syntax and control and data structures that everyday programmers have at their disposal. Large-scale features are those which help provide solutions to the problems inherent in large-scale software projects (e.g., design, maintenance, etc.).

The features listed in Appendix A are derived from the application characteristics which have been identified and include small- and large-scale features. This list illustrates strong application requirements for extensive data structure manipulation (structure definition, structure passing, etc.) and the modular design of applications (cross module checking, data hiding, separate compilation, etc.). Practical features such as exception handling and inter-language communication are also included.

Some languages supplement explicit features by providing preprocessors. Most preprocessors effect preliminary computation or organization by means of special commands which would otherwise be interpreted as comments and/or be expanded into pre-determined code sequences in the native language. For example, the MORTRAN preprocessor [9] expands non-native FORTRAN control structures into FORTRAN statement sequences which emulate the action of those structures.

The generation of a features list must be done very carefully so as not to include language-related topics that are not truly features. For example, this list does not

include vector processing capabilities or reentrancy.

Writing vectorizable program code is an important topic which is presently receiving considerable attention with respect to HEP applications. However, vectorization is a hardware and/or compiler feature rather than a language feature. On the other hand, HEP applications show a considerable degree of data independence which can be potentially exploited with parallel execution. Therefore, language features which address parallel and/or concurrent processing are applicable. The ability to generate reentrant code is a property of a compiler and operating system, not of a language.

Given these very well-defined system specifications, it is assumed that the specific features of the object-oriented, functional and logic programming languages are unsuitable for these applications. Therefore, this discussion is limited to "mainstream" procedural scientific programming languages. For the purposes of example, the candidate languages include Ada, C, FORTRAN 77, Pascal and PL/I (presented in alphabetic order as to illustrate no preference). FORTRAN 8x, as defined in its second draft standard [4], is also included.

For completeness, and where possible, the comparison is taken not with respect to either the ANSI or ISO standard, but with respect to the actual implementations on applicable target machines. An example of the importance of this apparent compromise can be illustrated in the examination of Pascal features. Pascal as defined by the ISO or ANSI standards does not support separate compilation of modules. Reliance to these standards would present an inaccurate view of the language in that most real implementations have been supplemented to support modular compilation.

Columns 2 through 7 of the table in Appendix A presents a comparison of these candidate languages considering the defined desirable features. If the availability of a feature was unclear with respect to the available reference materials, "unknown" is specified. Additional information is provided by numbered-references.

3.3 Programming Language Fit in a HEP Application Environment

The reference compilers used in this evaluation were chosen to run on IBM and DEC mainframes. These two vendors certainly represent a substantial market share of the hardware used in HEP computing. Compilers for all of the candidate languages were available as DEC product offerings. The VSFORTRAN, VSPASCAL and PL/I product offerings from IBM were used. At the time of this evaluation a suitable Ada compiler was not available for IBM. The Waterloo C compiler from WATCOM was used on the IBM system.

3.3.1 Technical Issues

The following categories of technical issues have been identified:

- native language features defined via standards
- implementation-dependent language extensions
- language tools and techniques

3.3.1.1 Implementation-dependent Language Extensions

The number-referenced items in Appendix A indicate how selected vendors have supplemented and/or modified specific features of the languages or provided language extensions. The references to DEC and IBM in these notes refer to the language compilers used on these machines, not necessarily to the compiler products of these vendors. (The specific reference compilers were cited earlier.) Some extensions, while providing the same functionality, may not be consistent in their implementation. If applicable, a standard's adherence with respect to such extensions is also cited.

3.3.1.2 Language Tools and Techniques

Appendix A also references relevant tools and programming techniques. The tools of particular interest are those which has been specifically developed to support HEP applications (e.g., ZEBRA and MORTRAN).

ZEBRA [7] is a dynamic data structure and memory manager developed at CERN. It allows the management of large amounts of data in computer store by providing the functions required to construct a logical graph of the data and their interrelations. ZEBRA is a very large project, having far reaching implications for the whole programming environment. Data managed by ZEBRA are stored in FORTRAN common blocks ("stores"). Each store can be subdivided into up to 20 "divisions" Relations between the basic units of data, or "banks", are expressed by attaching a structural significance to part of a bank. A bank is accessed by specifying its address in a given store. Such addresses (called "links") are kept inside the banks or in "link areas" inside a common block.

MORTRAN is a structured language macro-to-FORTRAN processor developed at SLAC to provide logical structures otherwise missing in FORTRAN. MORTRAN programs consist of MORTRAN statements interspersed with FORTRAN statements. Current versions of MORTRAN support the FORTRAN 66 and FORTRAN 77 language standards.

LINT (sometimes written as "lint") is a separate version of the C compiler modified as a program verifier. It does not generate code, but instead checks on a variety of program attributes which can be analyzed at compile and load time. Included are checks for potential portability problems, data types, unreachable statements, unreferenced variables, function arguments and returned values. LINT is part of the Programmer's Development kit and usually comes standard with C compilers.

3.3.1.3 Summary of Technical Issues

The information presented in Appendix A must be evaluated with care. It would be misleading to count the numbers of "yeses" and "noes" in order to reach some conclusion. Such a count would assume an equal weighting (value) of each of the technical features. The information should, however, be used to reach preliminary conclusions based solely on technical evaluation:

- Ada has the broadest base of features of the languages being discussed. It has an extensive range of array definitions, and permits subprograms to use arbitrarily sized arrays. Ada does *not* permit the user to define the internal repre-

sentation needed for multi-dimensional arrays [26]. Intrinsic array operations are *not* defined within the language, but viewed as an extension in an Ada package if required. Ada offers extended capabilities in the area of bit manipulation on the order of enumerated types. Real data types are defined within the language and support for extended precision and complex arithmetic can be provided through packages in a reasonable way, since overloading of operators allows the mathematical syntax to be preserved. However, Ada does not recognize the need for at least two (single and extended precision) structurally different floating point arithmetics[26].

Ada was obviously designed with conscious emphasis on sound software engineering. It has the best ability of the candidate languages to hide data and routines. It is especially strong in demanding a clarity of exposition of the data and routines which may be imported to, and exported from, any program unit by providing cross-module checking. Ada provides a wide range of I/O facilities though a set of predefined packages, such as DIRECT_IO, TEXT_IO, or LOW_LEVEL_IO. The **with** and **use** constructs can be used to select the appropriate set. Ada offers aids for concurrent computation. This is of little consequence for HEP off-line processing, but could be a major aid for on-line processing. The technical features of Ada with respect to HEP are discussed in greater detail in [28].

- C is a language with functionality similar to Pascal, but it is much less strict in the consistency checking of program units. LINT must be used to achieve the kind of cross module checking that Ada and Pascal offer. Data hiding is available at the same level as Ada. C defines I/O through a standard library (library stdio).

  C defines real and double precision data types in the language, and support for complex arithmetic can be provided using the typedef and struct facilities, but without retaining the mathematical syntax. C manages fairly well in the area of bit manipulation. Bit fields of variable widths can be incorporated into user-defined structures, and a pointer type can be used to locate the structure at a given address in memory. Sets are not defined. Enumerated types are not defined in Kernighan and Ritchie [17] but are in the ANSI draft standard [2]. The language provides a very useful preprocessor for performing such operations as macro definition and library inclusion.

- FORTRAN provides reasonable support for array handling and manipulation. It is the only one of the candidate languages which defines real, double precision and complex data types as part of the language. FORTRAN contains a very clear definition for formatted and unformatted I/O to terminal, printer and mass storage. However, it is clear that FORTRAN 77 lacks a great deal of the functionality that the other languages offer, and little type checking and no cross module checking. There is no data hiding beyond local variables. In realistic HEP applications, FORTRAN 77 must be supplemented by additional software tools. At CERN, the whole concept of ZEBRA is a manifestation of one of FORTRAN 77's needs (i.e., data structure manipulation). FORTRAN 8x has directly addressed many of these deficiencies of FORTRAN 77.

- Pascal, as implemented by both DEC and IBM, is very similar to Ada from the stand-point of computing itself. It offers the ability to hide data and code at the same level and cross module checking. It does not offer mul-

ti-programming as part of the language. Pascal provides rather simplistic I/O facilities in the Jensen and Wirth definition [16] which are not as comprehensive as those of FORTRAN and do not provide for unformatted (binary) data. Extended I/O facilities are common in implementation-dependent compilers.

Extensions in the area of array manipulation alleviate many of the problems which exist in the Jensen and Wirth definition. Bit manipulation is possible in an arcane way using sets that is poorly documented. Real data types are defined in the language, but double precision and complex must be emulated in user-defined data structures.

- PL/I is a very rich language in terms of its constructs, nearly on a par with Ada. It offers data hiding at the same level as Pascal. Like Ada, PL/I offers a language feature for exception handling. It offers rather little, however, in terms of type checking and no cross module checking. According to the language definition, concurrent computation is part of the language. However, this feature was not available in the IBM and DEC compilers used.

  Complex and double precision data types are defined as part of the language. Bit functions are also supported. Sets and enumerated types are not defined in the language. PL/I does provide a very powerful preprocessor which provides such capabilities as including text from an external library, conditional compilation of sections of the source program, macro development and variable name replacement.

The technical issues alone would appear to suggest the suitability of any of the candidate languages over FORTRAN. However, it is clear that this would obviously not be a realistic conclusion. From a historical perspective no programming language evaluation could be conducted with the prospect of completely replacing FORTRAN in HEP applications. This is substantiated in the evaluation of practical issues.

3.3.2 Practical Issues

The following categories of practical issues relevant to collaborative HEP software applications have been identified:

- historical role of FORTRAN
- portability
- inter-language Communication
- maintainability
- language standardization.

3.3.2.1 Historical Role of FORTRAN

The widespread adoption of FORTRAN by the physics community probably came about because it was the best approximation to the general purpose language, capable of abstracting most of the computer-oriented ideas that one physicist wished

to express to another (i.e., the *lingua franca*). Another possible factor is the fact that FORTRAN was the first compiled language and that it was developed specifically with the abstraction of mathematical expressions in mind (FORmula TRANslation).

Millions of lines of program code written in FORTRAN are a valuable foundation of the role of computing in high energy physics. Much of this code has proven itself over the course of many years to be accurate, reliable, efficient and flexible enough to be used as the requirements of experiments and even as physics itself have changed. Concurrently, the shortcomings of the language have led to the production of software libraries that alleviate many of its defects, and represent a huge investment in accumulated expertise.

From a historical perspective it is more realistic to evaluate programming languages in the HEP environment in such a way as to complement FORTRAN applications. To *fit* Ada, C, Pascal or PL/I into a HEP application of any consequence will depend on the ability of that language to communicate effectively with modules, libraries and other program units which were developed in FORTRAN.

It is an established fact that software developers tend to favor their first programming language even to the extent of applying the style or structures of this language when they program in other languages. This fact raises a potentially serious staff expertise issue. Staff who have learned programming with *modern* languages (perhaps post-1975), would have little or no difficulty adapting to a choice of any of the candidate languages. On the other hand, older staff may find themselves writing FORTRAN-style programs in other languages and losing the advantages that the newer language may provide.

3.3.2.2 Portability

An ideal (strictly standard) programming language should operate independently of the hardware and the operating system within which it functions. This concept allows the programmer the maximum degree of independence and permits the greater functionality allowed by the portability of software. For large computer manufacturers who support multiple operating systems and hardware lines, this independence provides the potential for a common interface.

For software to be shared effectively, it must be portable, that is, it must be capable of operating in environments other than the one in which it was developed. This includes different machines and therefore rules out the use of assembly languages except through interpretation. This has the disadvantage that there is considerable overhead associated with simulating one computer on another. Thus, the portability of software written in a high level language depends upon the availability of the appropriate compilers on the target machine or at least on the existence of a compiler that is itself portable and which can be moved easily to new machines at a cost far less than that required to produce it initially. Moreover, for portability to be successful, a common subset (or dialect) of the high level language must exist among the compilers.

Code portability is one of the major practical issues in a HEP experimental environment. Experiment collaborators are literally "seconds away" from one another via network. Estimates indicate that as much as two-thirds of the data processing involving CERN experiments is done at collaborating institutions. Therefore programs, subprograms, libraries, etc. are easy and necessary to share. Many of the

advantages of a collaborative environment would be lost if portability presented a major problem. Codes should run with minimum (or no) modification on all collaborative systems. Code *must* yield the same results on all systems.

Code sharing between collaborations is not the only interest in portability. Williams [29] and Zanella [32] see a continuing trend of downward migration of analysis code to "computing farms" of emulators and embedded systems. Specific language evaluations have been done by Williams for these applications.

Another dimension of portability/code compatibility has arisen beyond that of physics collaboration sharing. The exploding interest in computational physics has led to computer programs being distributed widely through professional journals. In the past if an article used program code, it was mainly for illustrative purposes. It appears that the rationalization is now for the widespread dissemination of software. It is quite likely that in the very near future physics journals may follow the trend of popular computing magazines by including programs on some form of computer medium.

3.3.2.3 Inter-language Communication

It is not uncommon in the evolution of a software system to want to program a new application in one programming language while still maintaining the use of existing libraries that have been programmed in a different language. In most cases the recoding of existing software solely for compatibility cannot be cost justified. This concept of *inter-language communication* has played a very important historical role. Subprograms and libraries programmed in assembler language offer greater efficiency to applications programs written in higher level languages

The increasing demand for program portability across multiple computer systems has pushed the demand for inter-language communication to the higher level language level from the assembly language level. Separate compilation of program modules allows for compatibility at the source code level if the module interfaces are precisely defined. The primary compatibility issues at these interfaces are:

- data type conventions – it is imperative that data type matching occurs with parameters passed to modules
- array conventions – it is necessary to know how multi-dimensional arrays are mapped into memory by the called and calling languages.
- calling conventions – it is necessary to know how parameters are passed to modules, how values are returned how register and memory management is handled.

A number of these concerns *within a particular language* can be addressed by that language's ability to perform cross module checking, data type compatibility checking, etc. This is also an area of concern in all software engineering methodologies.

Effective inter-language communication deletes the need to "re-invent the wheel" and allows the capability to integrate the old and the new in a satisfactory manner. In a HEP environment, this is a measure of the *fit* to existing FORTRAN libraries, graphics and database facilities.

The effectiveness of inter-language communication in HEP applications has been demonstrated in isolated circumstances. At SLAC, Lynch [20] wrote the program for ray tracing of particles through the magnetic and electric fields of the SLC positron target in VSPASCAL with an interface to HandyPak for histogramming. In addition, all analysis of the measured magnetic field for the SLD magnet was done in Pascal with an interface to MINUIT [15] for the final fitting.

The issue of inter-language communication also addresses the ability of programs to interact with the operating system. This is an implementation-dependent feature related to the overall issue of language bindings. There is presently standardization activity in this area [13].

3.3.2.4 Language Standardization

One of the premises of this evaluation was that language features would not be solely based upon the definitions in the ANSI or ISO standards (if they exist), but with respect to the actual implementations on applicable target machines. While this premise is useful in the short run, language standardization activity must be considered for projects with lengthy life cycles.

The technical direction assumed by the standardization activity of an existing programming language is of critical importance. Standards activities should attempt to preserve investments in software written in the language and to create new standards with as high a degree of compatibility as possible with previous standards. At issue is the question of *object code compatibility* and *source code compatibility*.

For example, it is instructional to note CERN's experience in the transition from FORTRAN 66 to FORTRAN 77. Since the old standard was almost a subset of the new, source code compatibility was fairly automatic. With object code, however, the one possible problem was with character string data. Prior to FORTRAN 77, character data was often handled by reading it into integer arrays. The 77 version provided a CHARACTER data type. The compatibility difficulty arose when a program compiled for FORTRAN 77 wished to call a subroutine compiled under the earlier standard, and to pass a character string constant as an argument. Had FORTRAN 77 implemented character strings as arrays of one-byte integers there would have been no problem. As a result, the documentation for the CERN Program Library (widely used in the HEP community) contains the following caveat:

> As regards language conversion of existing FORTRAN IV routines to FORTRAN 77 only a purely mechanical conversion to allow compilation without fatal errors has been performed. No attempt has been made to rewrite using FORTRAN 77 constructs, in particular no 'CHARACTER' type variables have been introduced. Where a FORTRAN IV construct gives a warning message but compiles correctly no change was made. Where a FORTRAN 77 construct was necessary but also works in FORTRAN IV the change was applied to both language versions.

3.3.2.5 Maintainability

Portability is really just one aspect of the wider problem of maintainability. Martin [21] defines the necessity for maintainability to:

- correct errors and design defects
- improve the design
- convert programs so that different hardware, software, system features, telecommunications facilities, and so on, can be used
- interface the program to other programs
- make changes in files or data bases
- make enhancements or necessary changes to the applications.

The classical problem which occurs with respect to maintainability is when:

- a program's logic is so complicated or uses so many "tricks" that even the author cannot remember how it was supposed to work
- there are programming language limitations (limited logical constructs, short variable names, etc.)
- there is poor documentation.

In the worst case such programs may become impossible to maintain and eventually have to be abandoned even though they originally performed exactly as required.

Preprocessors may provide a "stopgap" solution to the language problem by providing additional features. However, such a solution burdens the programmer with having to learn the features of the preprocessor and introduces the risk of inefficiency and error due to the preprocessing step.

As a result, Martin [21] suggests using the *highest level language possible* and software tools such as source code maintenance/management utilities.

3.3.2.6 Summary of Practical Issues

- Because of its conscious emphasis on software engineering, Ada, in its definition addresses the identified practical issues quite well. It is standardized [1] very strictly by the United States Department of Defense. It is claimed that neither supersets nor subsets of Ada will be permitted, and that all products claiming to be Ada-related will have to be periodically resubmitted for testing to insure that they conform to the specifications of the standard. This requirement addresses the issue of portability quite globally as long as there are suitable Ada compilers and tools for a full range of computers. The FORTRAN interface and the more general issue of inter-language communication are handled in the definition by the INTERFACE PRAGMA. However, inclusion of this pragma is not a compiler requirement and not included in compiler validation. The portability of language extensions provided in Ada packages must be

handled on an individual basis. Specific tools for the maintenance of Ada code are provided in the APSE (Ada Programming Support Environment).

- The *de facto* standard of C is the appendix of the Kernighan and Ritchie book. Most C compilers are based upon and adherent to this reference. It has, however, often been criticized for its terseness. An ANSI Draft Standard for C has been published [2] which addresses the missing or controversial issues of Kernighan and Ritchie.

  The C compilers evaluated contain facilities for inter-language communication. LINT and MAKE (contained in the Program Developer's kit) are very useful and powerful tools. LINT can be used to check for program portability. MAKE is a source code maintenance tool. C is one of the languages which has been targeted by both DEC and IBM [12] for consistency across their hardware and operating system lines.

- FORTRAN is the strongest of the candidate languages currently dealing with the identified practical issues. This is result of early identification of these issues and considerable and continuing work. The standard for FORTRAN 77 is ANSI X3.9-1978 [3]. The previous version of the standard (FORTRAN 66) was ANSI X3.9-1966. There was little, if any, participation by the HEP community in the definition of either of these standards. There is, however, considerable participation by the HEP community in the definition of FORTRAN 8x.

  Portability has long been an issue with FORTRAN-related tools. The FORTRAN libraries and facilities such as ZEBRA are available on a wide variety of machines. Maintenance tools are required for the generation of program versions for different computers, to enable many users to develop a single program, and to enable code to be transported readily from machine to machine. Here, a CERN-produced product, PATCHY, has found applications. A commercial product, HISTORIAN PLUS, is a portable source code management system that is used in some HEP laboratories in Europe [27].

  The MORTRAN preprocessor has been quite popular among SLAC collaborators because of the portability it affords. The output of MORTRAN is "vanilla" FORTRAN which should be perfectly portable, but not always the most efficient. The maintainability of MORTRAN is a separate, though related, issue.

  FORTRAN contains support of inter-communication defined within the language. It is also one of the languages which has been targeted by both DEC and IBM for consistency across their hardware and operating system lines.

- Pascal has some of the software engineering features of Ada, but lacks adequate standardization. Easy portability of Pascal code is jeopardized by implementation-dependent extensions added by most of the compiler vendors to supplement shortcomings of the original language definition [16] prior to language standardization [5]. The language is presently defined by the ANSI X3.97-1983 standard which includes many of these extensions. Pascal contains, in the language definition, an interface to FORTRAN and assembly language.

- PL/I is defined by the American National Standard PL/I [6]. Despite its advanced features and long-term availability, PL/I does not seem to have gen-

erated considerable interest in the HEP community. Specifically, its functional advantages over FORTRAN have not provided substantial reasons for acceptance. With the advent of newer languages with comparable functionality, this attitude is unlikely to change. The PL/I DECLARE and PROCEDURE statements contain parameters which allow for inter-communication to specific languages. IBM has not targeted the language for consistency across its hardware and operating systems lines despite its strong support of PL/I in the past.

Considerable interest has been generated by the growing development of software engineering methodologies. The application of such methodologies may provide solutions to some of the problems identified by both technical and practical software evaluations. Zanella [32] has expressed the opinion "...if forced to develop some long-lasting code, one should adopt some modern software engineering approach. About this issue it is often said that HEP has not kept in step with practices in the outside world, and has only recently become aware of the possibilities and problems associated with the production and maintenance of codes to be distributed to many users."

A review of software engineering methodologies conducted by James [14] indicates that software engineering techniques may provide solutions to a number of the problems arising from the practical issues (portability, maintainability, etc.) of physics project software. James even suggests that physicists are attracted by the structure and techniques (boxology, bubble charts) of the various methods.

Yourdon (one of the major proponents of software engineering) [31] simply states that such approaches allow programmers to "...identify the major functions to be accomplished and then...proceed from there to an identification of the lesser functions that derive from the major ones." This description of the software engineering process illustrates as its goal a top-down hierarchical description of a software project into independent program modules (functions).

The dependence of HEP code on subprogram libraries has made it inherently modular with well-defined interfaces between the modules. The portability and maintainability of these modules/libraries has long been a recognized concern. The opportunities to reduce system cost and fault rate by using (or re-using as the case may be) these existing modules should be recognized as a part of the requirements analysis and design activity for any new HEP application. They can then be treated in the logical design and coding step as the proverbial *black boxes* or *program stubs*.

Logical design and coding can then concentrate on the requirements of the new program code to be included in the project design. If the interface requirements to the existing code can be satisfied, then the design and coding of this new code is not restricted by the characteristics of the existing code. In other words, the new code can be generated using any programming language which supports the hierarchical modular design of the project and can support the interfaces to other program modules. This includes those languages which have the features of separate compilation of subprograms and strong inter-module and/or inter-language communication capabilities. The portability and maintainability of the new code can be considered independently of that of the existing (presumably maintainable and portable) code.

Such a structure would allow the system to grow and/or be maintained and modified at a manageable rate. The modular design of the old (and new code) would allow it be modified (or re-written) without major impact on the performance of the entire system. Testing of new modules and additional functionality is easily

allowed.

The paradigms in the various software engineering methodologies (boxologies, bubble charts) allow the management of these processes. The systems can be defined as to whether they are driven by data, processes or facilities, but the end result is basically the same.

What software engineering promises to hold for HEP is that it does address the practical issues which exist in the large HEP software projects. It also defines how the experience embodied in existing FORTRAN libraries is compatible with the expertise of the young physicists who have grown up with new technology and have learned programming and computational physics with programming languages and philosophies which reflect that technology.

3.4 Growth and Future Development

Zanella [32] describes a trend called "search for quality of life" which aims at a more comfortable computer environment in general. The important factors contributing now are things such as: quick response times, adequate and properly backed up file space, user friendly full-screen editors, *a choice of programming languages* and special debugging tools, good graphics capabilities, easily accessible DBMS, advanced document preparation facilities, electronic mail, etc. This applies to all systems, whether central or decentralized. For corporate computing this sounds like the complete "office system." For HEP it provides the possibility of a *unified physics computing environment.*

3.4.1 Hardware

There seems little doubt that HEP data analysis will, in the future, move downward in a distributed, decentralized manner from the current mainframes. It appears that there will be an increased use in personal computers, desktop mini-computers and workstations. *Computing at CERN in the 1990s* [*8*] has cited a goal of having "a personal workstation for every physicist and engineer at CERN."

Languages/programs must run on these machines with the same results. Languages, libraries and capability must be available on these machines. It is fully expected that high-speed, double-precision floating-point co-processors or array processors and expanded memory on these machines will make this possible.

At the same time, smaller machines are becoming more powerful, and the user-friendliness of their operating systems makes them increasingly attractive to physicists, who often prefer to carry out program development on a small computer belonging to their own group, rather than battling with a huge system running on a mainframe. Here we may see a move away from mainframes for development, with their use being reserved for batch number-crunching. The introduction of personal work stations attached to LANs can only accelerate that trend.

3.4.2 Software

In the area of software, user-friendly operating systems have already been touched on in the context of small computers. The other main trends in software are in the area of tools and languages. Programmers will increasingly expect to be able to work in an environment tailored to their own requirements, using tools which enable them to design, construct, test and maintain their software in a coherent fashion.

IBM [12] and other major software vendors have adopted a software strategy for the future which includes the goals:

- a common programming interface through which customers, independent software vendors and hardware manufacturers can develop applications that can be integrated with each other and ported to run in multiple environments;
- common communications support that will provide interconnection of systems and programs and cross-system data access;
- a common user access, including screen layout, menu presentation and selection techniques, keyboard layout and use and display options.

Similarly, committees such as ISO/TC97/SC22 are working on future guidelines for programming language bindings. Their goal is that standard bindings of some form should be developed for all standard system facilities that may be accessed from a standard programming language. Of particular interest are language bindings to systems facilities such as graphics and database services.

3.4.2.1 Document Preparation(Word/Text Processing)

Prominent in the *unified physics computing environment* is the capability for word/text processing. It is likely that physicists would want to do document preparation on the same systems where they do data analysis and software development. This would also facilitate the inclusion of program code/output/graphics into document text. Many of the issues which are applicable to programming language evaluation are also applicable to text and word processing systems (e.g., portability, maintainability). It is an increasing trend to develop these systems and supporting systems, such as graphics and printer drivers, in high order programming languages due to the portability which is afforded. Some systems allow user-developed macros to be written in these languages. The important role of *inter-language communication* can easily be seen in such cases.

It is possible that systems similar to Donald Knuth's W_EB ([18], [19]) will become more widely available. W_EB allows programmers to write code and documentation at the same time. Knuth feels that such practice leads to higher quality program documentation. W_EB is quite simply implemented with a preprocessor to a Pascal-like language which yields Pascal and T_EX source files. (It is only fair to mention that Pascal was chosen by Knuth due to its accessibility.)

4. Conclusions

This comparison of programming languages for HEP applications has led me to the following conclusions.

CERN's assumption that "FORTRAN will remain the main language for HEP code well into the 1990s" [8] is a valid, but conservative, one. There is simply too great an investment in existing HEP software of FORTRAN-based scientific subroutine libraries and support/analysis facilities. There is also the considerable investment in expertise which has resulted from the long association between FORTRAN and the HEP community. It is important that these investments be protected.

It is a sound strategy that "CERN and HEP should periodically review developments in languages." However, I believe that HEP should not just be content to review developments in other programming languages, but should also encourage software development in other languages *if it can be proven to be a sound software design decision.* The candidate languages evaluated in this work are all of sound design and strongly supported. This study indicates that they all satisfy, to varying degrees, the present and anticipated future computing requirements of HEP software projects. They are well-rooted in the current mainstream of data processing worldwide. Their capabilities can and should be utilized.

Zanella [32] has said " If HEP wishes to keep to its level of achievement, credibility and excellence, then it needs an injection of bright young computer-wise scientists and engineers." This means that HEP cannot become "an island." HEP applications must be able to utilize "state of the art" facilities in all areas of applicability including data processing. HEP must be able to take advantage of the technological advancements in other arenas of science and engineering. Many of these advancements are occurring in fields which are presently *not software compatible* with HEP. Much of the work being done in embedded systems with Ada or telecommunications with C could be of great interest and applicability in HEP computing environments. The *unified physics computing environment* anticipated for the 1990s should be able to take full advantage of these facilities and the physicists and engineers of the 1990s should be able to take full advantage of their *unified physics computing environment.*

Computing at CERN in the 1990s [8] states that "The CERN or HEP community should periodically review developments in languages...to *add weight to their input to FORTRAN standardization committees."* FORTRAN 8x holds great promise as a programming language, perhaps, as some believe, it will be the most powerful of the procedural languages. Many of the defects in FORTRAN 77 discussed in this paper (array handling, data hiding, etc.) have been/are being dealt with by X3J3, the FORTRAN 8x standards committee [24], [25]. Many of the software engineering considerations of Ada which have been discussed promise to be in FORTRAN 8x.

The problems of the FORTRAN 8x standards committee have been well-documented in arenas such as the ACM FORTRAN Forum. When a standard has been produced by the committee, there is no indication of how widely it will be accepted and how long it will take for reliable compilers to reach the market. The migration of present HEP applications to FORTRAN 8x promises to be a time-consuming, non-trivial task. Features which have been emulated in the past will now be directly supported by the language. A case in point are the *many* appli-

cations which rely upon the data structure manipulation facilities of ZEBRA. Such migration should necessarily be done at the source code level.

HEP software developers should not be forced to wait for FORTRAN 8x when other programming languages with considerably more desirable features than FORTRAN 77 are available now. The art of software engineering can teach HEP how to integrate those languages or facilities in those languages into HEP applications. The examples in SLC and SLD have proven that this is feasible. The scientific personnel entering the HEP field in the 1990s will be versed in these languages and techniques. When (and if) FORTRAN 8x becomes widely available, it too will be compatible with these designs so that there will have been no loss of development time and/or effort.

5. Acknowledgement

I would like to acknowledge the help of many colleagues in the DD Division of CERN and the SCS Division of SLAC who made contributions to this paper and to my way of thinking. Mike Metcalf of CERN was especially helpful with his comments and information regarding the FORTRAN 8x effort. I am also particularly indebted to Harvey Lynch and Jim Russell of SLAC for the work they have done in the evaluation of programming languages.

Appendix A

Detailed Language Features Comparison

	Ada	C	F77	F8x	Pascal	PL/I
Address arithmetic	(1)	Yes	No	No	Yes	Yes
Arbitrary array bounds	Yes	(2)	Yes	Yes	Yes	Yes
Arbitrary function return value	Yes	(3)	No	(4)	Yes	(5)
Argument by name or position	Yes	No	No	Yes	No	No
Array bound checking	Yes	No	(6)	No	Yes	Yes
Array of structures	Yes	Yes	(7)	Yes	Yes	Yes
Array operations	(8)	No	No	Yes	(9)	Yes
Binary I/O	Yes	Yes	Yes	Yes	No	Yes
Bit logic	(10)	Yes	Yes	Yes	(11)	Yes
Call by reference	Yes	Yes	Yes	Yes	Yes	Yes
Call by value	Yes	Yes	No	(12)	Yes	No
Complex variables	Yes	(13)	Yes	Yes	(14)	Yes
Concurrent computation	Yes	No	No	No	No	(15)
Cross module argument checking	Yes	(16)	No	(17)	Yes	No
Data hiding	Yes	Yes	No	Yes	Yes	(18)
Descriptive variable names	Yes	Yes	(19)	Yes	Yes	Yes
Double precision	Yes	Yes	Yes	Yes	(20)	Yes
Dynamic memory	Yes	Yes	(21)	Yes	Yes	Yes
Enumerated data types	Yes	(22)	No	No	Yes	No
Exception handling	Yes	No	(23)	(24)	(25)	Yes
Formatted I/O	Yes	Yes	Yes	Yes	Yes	Yes
Inhomogeneous data structures	Yes	Yes	(26)	Yes	Yes	Yes
Inter-language Communication	(27)	(28)	(29)	(29)	(30)	(31)
Local procedures	Yes	No	No	Yes	Yes	Yes
Multiple subprogram entries	(32)	No	Yes	Yes	No	Yes
Pass arbitrary matrix	(33)	(34)	Yes	Yes	(35)	Yes
Pass arbitrary 1-dimensional array	Yes	(36)	Yes	Yes	(37)	Yes
Pass structure to subprogram	Yes	Yes	(38)	Yes	Yes	Yes
Pointers to functions	(39)	Yes	No	No	No	No
Pointers to variables	Yes	Yes	(40)	Yes	Yes	Yes
Preprocessor	(41)	Yes	(42)	No	(43)	Yes
Recursion	Yes	Yes	No	Yes	Yes	Yes
Routine hiding	Yes	Yes	No	Yes	Yes	(44)
Sets	Yes	No	No	No	Yes	No
Static variables	Yes	Yes	Yes	Yes	(45)	Yes
Strings	Yes	Yes	Yes	(46)	(47)	Yes
Strong type checking	Yes	(48)	(49)	(49)	Yes	No
Subprogram argument checking	Yes	(50)	No	(17)	Yes	No
User generic functions	Yes	No	No	Yes	No	No
Variable equivalence	(51)	(52)	Yes	Yes	(53)	Yes
Variable initialization	Yes	Yes	Yes	Yes	(54)	Yes
Variable range checking	Yes	No	No	No	Yes	No

(1) Unknown
(2) Low = 0

(3) In Kernighan & Ritchie, only simple variables or pointers;
ANSI draft standard also includes structures;
(4) User generic functions may be written
(5) Scalars, bit strings, string, entries, pointer, some expressions
(6) DEC – Yes; IBM – with IAD facility
(7) DEC – Yes; IBM – No
(8) Use operator overloading (external in a package)
(9) Can assign an array to another, but no operations on arrays as units
(10) Unknown
(11) Implementation dependent
(12) The INTENT of a dummy argument may be specified
(13) Can use typedef and struct, but without mathematical syntax
(14) No, must be emulated with records
(15) Defined in the language definition; not implemented by DEC or IBM
(16) No, necessary to use LINT
(17) Yes, if INTERFACE blocks are used
(18) No, only in local procedures
(19) DEC – Yes; IBM – No (only 8 characters allowed)
(20) No, must be emulated
(21) No, use ZEBRA
(22) No in Kernighan & Ritchie; Yes in the ANSI draft standard
(23) Implementation dependent
(24) Yes, for I/O
(25) Implementation dependent
(26) DEC – Yes; IBM – No, use ZEBRA
(27) Defined via INTERFACE PRAGMA; not always implemented
(28) DEC – Yes; IBM – Yes; standard definition – unknown
(29) EXTERNAL declaration
(30) Defined via EXTERNAL and FORTRAN declarations; implementation dependent
(31) Yes, with parameters of the DECLARATION and PROCEDURE statements and only with specific languages
(32) Unknown
(33) Unknown
(34) Pass array of pointers to arrays; both must start at 0
(35) Pass array of pointers to arrays; turn off range checking
(36) Lower bound must be 0; no bound checking possible
(37) DEC(ISO) – Yes; IBM(ANSI) – No; turn off range checking
(38) DEC – Yes; IBM – No
(39) Not in the sense of C; pass routine at FORTRAN level
(40) No, use ZEBRA
(41) Conditional compilation, declaration import, and simple lexical substitution of constants with arithmetic are part of the language
(42) Yes, MORTRAN is one example
(43) Substitution of constants with arithmetic is supported by DEC and IBM
(44) No, only as local procedures
(45) Implementation dependent
(46) Can be coded as a MODULE
(47) Implementation dependent
(48) Yes for simple variables, no for routines
(49) If IMPLICIT NONE is used
(50) No, must use LINT
(51) Unchecked_conversion or variant record

(52) Variant record or union
(53) Variant record or union
(54) Implementation dependent

Bibliography

1. American National Standards Institute, Inc., *American National Standard Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A-1983 (1983).

2. American National Standards Institute, Inc., *Draft Proposed American National Standard for Information Systems – Programming Language C*, ANSI X3J11/87-221 (November 9, 1987).

3. American National Standards Institute, Inc., *American National Standard for Information Systems – Programming Language FORTRAN*, ANSI X3.9-1978 (1978).

4. American National Standards Institute, Inc., *Proposed FORTRAN Standard*, S8 X3.9-198x (1988).

5. American National Standards Institute, Inc., *American National Standard Pascal Computer Programming Language*, ANSI/IEEE770X3.97-1983 (1983).

6. American National Standards Institute, Inc., *American National Standard Programming Language PL/I*, ANSI X3.53-1976 (1976).

7. Brun, R., Goossens M., and Zoll, J., *ZEBRA Users Guide*, CERN Program Library Q100, CERN DD/EE/85-6 (1985).

8. *Computing at CERN in the 1990s – Computing for Experiments.* 21 October 1988.

9. Cook, J., *MORTRAN3 User's Guide*, SLAC Computation Research Group Technical Memorandum 209, January 1983.

10. Cugini, J., *Selection and Use of General-Purpose Programming Languages*, NBS Special Publication 500-117, National Bureau of Standards, Gaithersburg, MD (1984).

11. Feuer, A., and Gehani, N., *Comparing and Assessing Programming Languages*, Prentice-Hall, Englewood Cliffs, NJ (1984).

12. IBM Systems Journal, *Introduction to Systems Application Architecture*, vol.27, no.3, 1988.

13. ISO/TC97/SC22, *Revised Working Draft on Guidelines For Language Bindings*, December, 1988.

14. James, F., *Do Physicists Need Software Engineering?*, Computer Physics Communications 41 (1986) pp.205-216

15. James, F., and Roos, M., *MINUIT – Function Minimization and Error Analysis*, CERN Program Library D506, 1988.07.25.

16. Jensen, K., and Wirth, N., *Pascal User Manual and Report*, Springer-Verlag, New York, NY (1974).

17. Kernighan, B., and Ritchie, D., *The C Programming Language*, Prentice-Hall, Englewood Cliffs, NJ (1978).

18. Knuth, D., *Literate Programming.* SHARE 60, San Francisco, CA, February, 1983.

19. Knuth, D., *T_EX : The Program*, Addison-Wesley, Reading, MA (1986).

20. Lynch, H., private communication.

21. Martin, J., and McClure, C., *Software Maintenance – The Problem and Its Solutions*, Prentice-Hall, Englewood Cliffs, NJ (1983).

22. Metcalf, M. *The Role of Computing in High Energy Physics.* CERN DD/83/15, September 1983.

23. Metcalf, M. *Effective FORTRAN 77.* Oxford University Press, London (1985).

24. Metcalf, M. *FORTRAN 8x – the Emerging Standard.* CERN DD/87/1, January 1987.

25. Metcalf, M., and Reid, J., *FORTRAN 8x Explained.* Oxford University Press, London (1987).

26. Morris, A., *Can Ada Replace FORTRAN for Numerical Computation?*, SIGPLAN Notices, vol.16, no.12 (1981) pp.10-13

27. OPCODE, Inc., *Historian Plus User's Manual*, Austin, TX (1988).

28. Russell, J. *Programming Languages: Time For A Change?.* Computing Physics Communications 45 (1987) pp.269-273

29. Williams, D.O. *Software and Languages for Microprocessors.* CERN DD/85/27, November 1985.

30. Williams, D.O. *Language Requirements for Embedded Systems.* CERN DD/86/18, December 1986.

31. Yourdon, E. *Techniques of Program Structure and Design.* Prentice-Hall, Englewood Cliffs, NJ (1975).

32. Zanella, P. *Trends in Computing for HEP.* CERN DD/85/12, July 1985.