SLAC – PUB – 4258
March 1987
(E)

PROGRAMMING LANGUAGES: TIME FOR A CHANGE?*

J. J. RUSSELL
Stanford Linear Accelerator Center,
Stanford, California 94305

Must computer programs read like, well, computer programs? An overview of the software needs of the high-energy physics community and how modern languages can meet these needs is given. Using ADA as an example, the production of readable, efficient and maintainable code is shown to be directly supported by the language, integrating concepts such as top-down design, object-oriented programming and data encapsulation as a natural part of the language rather than as foreign ideas imposed on the language. Particular attention is paid to nontraditional aspects of a language and how these can help by providing an integrated support environment for all phases of the lifetime of the software.

Published in Computing Physics Communications 45(1987) pg. 269–273;
also Presented at the Computing in High Energy Physics Conference,
Asilomar, California, February 2-6, 1987

---

* Work supported by the Department of Energy, contracts DE–AC02–76ER02220 (Princeton) and DE–AC03–76SF00515 (SLAC).

† Address after March 1, 1988: Theoretical Physics Group, Tata Institute of Fundamental Research, Homi Bhabha Road, Bombay 400005, India.

1. INTRODUCTION

For the past twenty years the high-energy physics community has used basically the same programming language and software techniques. During that time, not only has the size of software projects increased greatly, but so has the scope. Many programs now have very little to do with formula translation. They are concerned with pattern recognition, synchronous and asychronous control functions, database query and management, data acquisition, etc. Those in the online world have been struggling for years with requirements the present environment is unable to satisfy. It is interesting to note that with the arrival of interactive analysis programs and parallel processing demands, the vocabulary (and problems) of asynchronous control and multitasking familiar to the online world is making its way into the batch-oriented language of the offline community.

To be sure, improvements in software engineering have been made, but these have been evolutionary rather than revolutionary—continually erecting new facades around a very old building. Contrast this with what has been done in electronics. During the same time period, electronics has progressed from discrete transistors to integrated circuits to VLSI technology to custom chips designed with the aid of silicon compilers. The use of electronics in experimental high-energy physics has paralleled the natural development of the field. The software effort in high-energy physics has lost contact with the mainstream.

The next generation of detectors promises to greatly increase the demands placed on its software. Will the answer to this increased complexity be to add yet another layer to cover up the inadequacies of the current environment? This paper suggests that a different tack be taken: examining, not what can be added to a programming language *ex post facto,* but what a programming language can contribute directly. To do this, it is desirable to step away from the current environment and reexamine the underlying needs and enumerate those features and tools which are useful for the production and maintainence of quality software. For definiteness, ADA, the programming language mandated by the United

States Department of Defense for its software, will be used as the vehicle to illustrate how a modern language can not only fulfill many of the usual requirements of a language but also help solve problems which have not traditionally been viewed as being within a language's domain.

2. DESIRABLE FEATURES OF A LANGUAGE

The success criteria of software has long been dominated by one measure: does it work. The achievement of other desirable features have been either sacrificed or ignored. The resulting situation is much like that of balancing a pencil on its point: yes it is functionally operational, but, please, nobody breathe. One sees this in many experimental groups where either no one knows how something works or the code is so patched with fixes that it is a house of cards waiting to fall.

Five additional and important features of software are:

1. Readability
2. Efficiency
3. Maintainability
4. Portability
5. Provision of an Integrated Support Environment.

Ideally the achievement of any one of these particular features should not come at the expense of another one. Unfortunately, reality forces compromises, with efficiency generally squaring off against the others. In the current environment, the only tool supplied by the language which speaks to these needs is the subroutine/function construct. This is not sufficient. While this construct can make the code more readable, maintainable and portable, it generally has a negative impact on efficiency and does nothing to help with the problems of providing an integrated support environment. Even in its domain, it is not always

the appropriate construct for achieving a goal. Not all programming problems can be solved by inventing a subroutine. The popularity of preprocessors is an admission of this fact.

2.1 READABILITY

The code must be readable and readable at different levels of detail. Most physicists have neither the time, desire nor need to understand each line of code in full detail. The language should not only allow one to implement the ideas and concepts needed to solve the problem, but also allow the implementation to be clean and direct. One need only look as far as attempts to provide linked lists and data structures in FORTRAN for a violation of this principle.

2.2 MAINTAINABILITY

Given that the code is initially functional and readable, maintainability is what keeps it in that state. The maintainence of a particular package is relatively easy if its sphere of influence is well-defined. A primary contributor to maintainence problems is the failure to cleanly define the interface. This usually happens by exposing the user of a package to unnecessary implementation details. This exposure is likely justified by the increased efficiency it affords. Again, the example of linked lists illustrates the point nicely. The operation of moving to the next list member may be implemented as a function call, but efficiency dictates otherwise. This ensures that detailed knowledge of moving from element to element is buried within user code, eventually becoming scattered beyond the control of anyone to change it.

An often overlooked area is the need to plan for a package's demise. This may seem like a strange requirement, but the maintainence of software packages long past their prime is a very real problem. Consider the future of current packages which implement pseudo-data structure capabilities within FORTRAN. What is to be their role when data structures are directly supported within the language?

Their growing irreversible entanglement with many other packages would seem to ensure their continued and otherwise unnecessary survival.

2.3 EFFICIENCY

There is no escaping the fact that code must be efficient. Inefficient code costs time and money. Many alternate languages have been dismissed because of their failure to achieve reasonable efficiency. One must ask if a language is intrinsically efficient, or if it includes features which are not only in and of themselves inefficient, but which affect the efficiency of code even when they are not used. For example, early specifications of PL/I suffered somewhat in this area. Care must be taken to separate the intrinsic efficiency of a language from a particular implementation's efficiency. While one can make a bad implementation of anything, the question is whether one can make a good implementation.

One must also beware of false efficiency. Gains achieved by efficiently generating code for routines needed only because of language restrictions are not real. The greatest gains in efficiency come, not from great optimization, but rather from great algorithms. The difference between unoptimized code and highly optimized code may be a factor of two. The gains realized by the choice and implementation of the correct algorithm are usually much higher. However, these algorithms generally demand thinking in more abstract terms and the implementation of more complex code, neither of which may be obvious or practical in a primitive language. The increased understanding of optimization techniques in recent years will also minimize potential losses.

The cost of efficiency must also be accessed. Does the quest for efficiency make the code unreadable and unmaintainable by exposing the user to implementation details? Does this exposure to the inner workings preclude switching to some even better algorithm in the future?

2.4 PORTABILITY

While it is certainly desirable that code work on more than one machine, the question is the price to achieve portability. Portability is not guaranteed merely by writing in a high level language. The cost comes either in terms of efficiency, when one carefully layers out all machine dependencies to a collection of subroutines and functions, or in readability and maintainability, when one uses a preprocessor to implement conditional compilation. A good language should minimize the number of machine dependencies and, when they inevitably arise, it should provide a mechanism which preserves the semantics of the subroutine style without the efficiency penalty.

2.5 INTEGRATED SUPPORT ENVIRONMENT

A programming language is not an isolated object. In large projects it must be surrounded with various support tools. The language should provide enough information so that all the following tools work smoothly.

1. Syntax Oriented/Directed Editors
2. Compiler
3. Linker
4. Symbolic Debugger
5. Document Generator
6. Code Maintainence Utility.

The advent of editors which understand the syntax of a language removes the sometimes tacit design goal of keeping a language terse. Since the programmer need no longer remember language constructs in full detail, they can be made verbose enough such that their meaning is clear and unambigious to both human readers and other computer programs. This latter catagory includes not only the compiler, which will benefit by being able to generate more informative and

accurate error messages, but also document generators and code maintainence utilities.

The more directly a language supports abstract concepts, the more directly the debugger can provide information about them. A user-implemented data structure facility is an excellent example of functionality being achieved so far outside the general context of the language that the debugger cannot supply a fraction of the power it should.

3. LANGUAGE FEATURES

A language consists of two parts, small- and large-scale features. Small-scale features have to do with the syntax and control and data structures the everyday programmer has at his disposal. Large-scale features are those which help provide solutions to the problems inherent in large-scale software projects. The reality of large projects has forced more and more time to be spent on the organization and maintainence of code. Beyond the ability to do separate compilations, large-scale features have not, until this time, been considered as part of a language.

ADA has been chosen as a representative modern language to illustrate what can be achieved entirely within a language. Since a complete discussion of ADA is outside the scope of this paper, the reader is encouraged to consult an introductory book on ADA [1,2].

3.1 SMALL-SCALE FEATURES

The small-scale features which ADA supports and are useful in high-energy experimental physics' code are:

1. Complete and Rich Set of Control Structures
2. Data Structures
3. Strong Typing

4. Support for Low Level Coding
5. Block Structuring
6. Dynamic Memory Allocation
7. Overloading
8. Pragmas
9. Exception Handling
10. Tasking Support
11. Generic Facility
12. Default Arguments and Passing by Name.

This list is only meant to enumerate and not explain. Only the issue of data structures is expanded on below because it so important and ADA's facilities are so complete. The true power of these features comes when used together. As an example, consider the combination of the generic facility, overloading and compiler pragmas. A compiler pragma is an instruction to the compiler which is embedded in the code. Simple pragmas may control the listing, while more complicated pragmas may define the interface to a routine written in a language other than ADA (see below). Overloading refers to the ability of an operator or the name of a procedure or function to be context sensitive. Overloading is not something completely new, e.g., the arthimetic operators in most programming languages are overloaded to allow the exact function to change depending on whether the operation is between two real numbers, or two integer numbers, or a real number and an integer. ADA merely makes this feature available to the user. The ability to write object-oriented code depends on overloading. The generic facility provides what amounts to a civilized macro facility. While generics may not do everything that a traditional macro processor may do, they do allow the construction of generic packages. Since the semantics of generics are very similar to that of an ordinary procedure, the programmer is not unduly burden by a completely different set of rules to implement and use a generic package.

Using these three features together, the ADA programmer could create a suite of generic sort routines. The 'variables' to such a generic sort package would be the method of sorting, the data type to be sorted and the procedures to compare and swap data elements. The use of the inline pragma would guarantee efficient comparison and swapping of the data elements, while the overloading feature would place the burden on the compiler, rather than the user, to select the proper sort for a given data type. The combination is powerful, producing efficient and maintainable code.

3.1.1 Data Structures

Data structures are one of the most important considerations in any software project. The need for flexible and efficiently implemented data structures is currently the largest single deficiency in FORTRAN. The already alluded to attempts to provide such a facility for use in a FORTRAN environment only serves to illustrate their importance by the lengths one is willing to go to provide them. ADA provides all the traditional support one would expect.

1. Ability to mix inhomogenous data types.
2. Ability to separate the definition of the data from the instantiation of the data, i.e., COMMONs do not satisfy this criteria.
3. Ability to define new data types.
4. Ability to define recursive data structures, i.e., be able to point to another instance of a data structure of the same or different type.
5. Ability to define enumerated types.

In addition, ADA goes beyond these to provide extra capability which is particularly useful in high-energy experimental physics' code:

1. Ability to have runtime defined lengths and structure through the use of discriminated records and variant records.

2. Construction of data types that can be mapped directly onto a preexisting structure, e.g., hardware device control block.

3. Construction of private data types, allowing the hiding implementation details.

3.2 Large-Scale Features

The most useful concept ADA brings to this area is that of packages. This is the single biggest difference between ADA and older languages. A package allows one to cleanly isolate the specification of a software component from the implementation of that component. This separation is usually physical, the two pieces going into two distinct files. Because it may be impractical to contain the entire implementation piece in one file, ADA has the concept of a subunit which allows the placement of a segment of the package in another file. The ability to define a private piece allows the implementer to hide details and expose only what is necessary. These concepts accomodate and support both top-down design, used for construction of large projects, and bottom-up design, used in the construction of library utility packages.

Because the user of a package must explicitly declare within the code his intention of using the package, a number of hidden beneficial side effects are realized:

1. The entire dependency tree is carried with the code allowing other utilities to make use of it.

2. The writer of a package, at the time of writing, defines exactly what it is he wishes to use, i.e., the correct use of the code is not left to the mercy of a magical link command to pick up the components the package may depend on.

3. Name conflicts are drastically reduced to just ensuring unique package names. A package member, whether it is a data structure, function or

procedure, can always be qualified with the package name to ensure uniqueness.

4. Strong type checking and currency can be implemented by the compiler since all the necessary information is present at compilation time.

Another very useful construct ADA brings is a formal prescription used to interface a routine written in another language to ADA. Certainly this is very important and useful when initially migrating to ADA from another language. Through the use of a pragma one can specify the language the routine is written and the exact method by which the arguments are passed. This can then be bundled in a package in such a way that the use of the routine is transparent, i.e., the user cannot tell whether the routine is written in ADA or some other language. This feature helps preserve the investment made in the existing large base of software.

4. SUMMARY

The changing requirements of software in high-energy physics justifies carefully examining alternate languages. In addition to supplying all of the existing functionality of FORTRAN, ADA can help solve problems inherent in large-scale software projects. This, together with its increased functionality and ability to interface with existing software, makes it a much better base on which to build new code while at the same time preserving the investment in existing code. Even though the language may not be a perfect match to the software needs of experimental high-energy physics computing, it appears sufficiently flexible and general that a careful working study of its use is merited.

5. REFERENCES

1. S. J. Young, *An Introduction to ADA,* Second Revised Edition, Ellis Norwood, 1984.

2. J. G. P. Barnes, *Programming in ADA,* Addison–Wesley, 1982.