CONF-801018--1
SAND 80-1051-C

Teaching Old Fortran Programmers New Tricks *

Abstract

For a number of valid reasons, Fortran remains in widespread use. It can be difficult to get long time Fortran programmers to accept the use of new software tools that are increasingly required to lower software costs. In order to gain acceptance for a new software tool, it is necessary for it to be easy to learn and use, as well as to provide new benefits. In the process of introducing the use of the Ratfor preprocessor for Fortran, a number of useful guidelines were defined for gaining the acceptance of any new software tool in an existing environment.

Bruce E. Wampler, PhD
Sandia National Laboratories
Division 1723
Albuquerque, NM 87185
(505) 844-8414

In the event that this paper is accepted, the author will attend COMPSAC '80.

* SAND80-1051C - Describes work performed at Sandia National Laboratories under D.O.E. contract DE-AC04-76DP00789.

eB

Teaching Old Fortran Programmers New Tricks *

Bruce E. Wampler
Sandia National Laboratories
Albuquerque, NM 87185

Introduction

The rising cost of computer software has made it essential to use state of the art software engineering practices to improve software productivity. Almost no one would disagree with the fact that the use of modern techniques of structured program design, preferably supported by a structured programming language, can reduce both initial software costs and long term maintenance costs.

While the ideal would call for the universal use of a structured programming language such as Pascal, Fortran is still the language most widely used in a large part of the computer software industry. Fortran has many qualities that keep it the one truly universal programming language available today. First and foremost, a Fortran compiler is available for virtually every computer made. Fortran is usually heavily supported by the major computer manufacturers, with its compiler often generating the most efficient code of any language processor available for a particular machine. In addition, Fortran usually interfaces to a large library of system support routines and utility

----

* SAND80-1051C - Describes work performed at Sandia National Laboratories under D.O.E. contract DE-AC04-76DP00789.

programs, as well as to assembly language programs.

There is another reason for Fortran's continued use: many programmers already know it, use it, and if not love it, at least see no reason to leave it. While most recent graduates from computer science programs have been heavily exposed to structured programming techniques and languages, there are a great many people already working in the software industry who "grew up" on Fortran. For these people, who have been "thinking" in Fortran for many years, the advanced control and data structures of a language like Pascal can be extremely difficult to learn. In addition, there can be a great deal of inertia involved and programmers are often reluctant to leave the security of the status quo.

Even though these and other factors have resulted in the continued use of Fortran, it simply does not contain the language features needed to support structured programming. Fortran 77 is not much of an improvement. The programmer still lacks a complete set of loop control structures, is limited by data types, and is forced to use a rigid and inconvenient statement layout format.

One solution to the problem is the use of a preprocessor that converts a language with structured control statements and pleasant cosmetics into standard Fortran. The good points of Fortran are retained, while the benefits of a structured language are also obtained. Since the underlying language is Fortran, it is also an easier (although still not trivial) task for long time Fortran programmers to learn the new language. Once of the best and most widely available preprocessors is Ratfor (Rational Fortran), originally developed by Kernighan and Plauger and described in their book Software Tools [1]. Ratfor provides structured control statements (if else, while, for, repeat, and do loops) as well as some other features that make the language easier to use (free form input, character string facilities, better comment conventions, define statement, and include file processing).

The author is currently involved in a large project at Sandia National Laboratories that is will link a large number of remote data gathering computers in different parts of the country to a central site for processing and display. While the use of a structured language like Pascal would be very desirable to use for software development, several factors dictated the use of Fortran: it was the best supported language on the computer used for initial development work, the ultimate target computer was unknown, and the other project members were long time Fortran users. Since Fortran is so unpleasant to use (at least for someone accustomed to a more structured language), Ratfor seemed to be a reasonable alternative.

While Ratfor has a number of advantages over Fortran, the other project members were not ready to instantly accept its use. It should be emphasized that the use of Ratfor depended solely on user acceptance in an informal environment, with no decrees from management to compel its use. The Ratfor preprocessor (as described in Software Tools and distributed by Addison Wesley Publishing) was designed with portability as a major goal and as result lacks a number of features needed for program development work in a professional production environment, such as complete error diagnostics and a clean interface to the existing operating system environment. It took a number of changes to the Ratfor preprocessor and the implementation of additional support tools before the other users were willing to accept Ratfor. The resulting Ratfor software development environment has turned out to be far superior to both Fortran and the original version of Ratfor.

Guidelines for gaining acceptance of a software tool

The rest of this paper describes the author's experiences in getting a new software tool accepted by users in an established environment. While these experiences describe the problems getting Fortran users to accept Ratfor, they actually reflect the basic human nature. Consequently, these experiences can be readily extended to the general situation of getting established users of any system to accept a new software tool. The following guidelines summarize these experiences.

1. It should be easy to learn.
2. It should conform to present system usage conventions.
3. It should interface with existing software: system services, utilities, libraries.
4. It should have adequate documentation and support tools.
5. It should run efficiently and generate efficient code.
6. It should be as flaw free as possible.
7. Its benefits should be great enough to justify leaving the current system.

Each of the above guidelines is discussed in greater detail in the following sections.

1. It should be easy to learn.

Recent graduates of computer science schools have received training in the use of structured programming languages and techniques. Most of these graduates have been taught to learn and master a new programming in a short time as a matter of course. Gaining acceptance for a new language or tool from this segment of programmers is a relatively easy task.

On the other hand, many of programmers working in industry today have not had this training (usually because of a lack of opportunity), and find learning a new language a difficult task. Many of these programmers have been "raised" on Fortran, and are used to solving problems in Fortran. After spending many years thinking of problems in terms of Fortran, it is often very difficult to change into the structured mode of thinking allowed by a language such as Pascal. And while the structured control constructs are difficult enough for Fortran programmers to learn, the rich data types and structures used by languages such as Pascal can be totally bewildering.

Faced with the double problem of learning new control constructs as well as complicated data type declarations and syntax, many Fortran programmers will give up before they have the chance to realize the full benefits provided by a modern language. In order for them to even try a new language, let alone accept it for full time use, it must be easy to learn. The learning process can be facilitated if much of their knowledge of Fortran can be applied directly to the new language.

2. It should conform to present system usage conventions.

Once the programmers have been convinced to try a new language, they should not be forced to also learn a new set of rules for using the language on the existing operating system. The new language should be usable in the same fashion to which the users have become accustomed when using

Fortran. They should be able to use the same file system, the same editors, and the same operating system command language syntax to compile a program. If listings and compilation diagnostics are produced by the new language, they should use the same conventions used by the Fortran compiler. If the new language comes as a stand alone system with its own editor and command syntax, there will be more obstacles to be overcome by the new user, and widespread acceptance is unlikely unless the new system gives an immediate and obvious improvement over the existing system. Usually however, programmers like with stick to familiar operating procedures, even if they are not as good as might be possible.

3. It should interface with existing software.

Typically, an operating system will provide the programmer with a number of system services to perform various functions such as obtaining system time, performing specialized I/O, and other such system dependent tasks. In addition, there are often other utility libraries, such as graphics packages, mathematical analysis routines, and sort utilities provided within the normal environment of the operating system. Most commonly, the programmer has two choices for interfacing with these utilities: assembly language or Fortran.

Fortran users are accustomed to these facilities and are not usually willing to give them up. In order for a new language to be accepted, it too should interface to these or equivalent services and libraries. This interface should be easy to use. Requirements for special declaration statements or use of indirect interface technique can lead to rejection of the entire tool or language, no matter how great the other benefits may be.

4. It should have adequate documentation and support tools.

While Fortran may not be the ideal problem solving language, at least the procedure for compiling, linking, and executing a Fortran program will usually be straightforward, and the techniques needed for performing that procedure are usually well documented and familiar from long use. Any new language or tool must be at least as easy to use as Fortran. Documentation required to use the language (language specification, user's manual, etc.) must be complete and useful. Incomplete, obscure, or inaccurate documentation can be worse than no documentation at all.

In addition to generating object code, Fortran compilers usually are able to produce very complete listing files which usually include a numbered listing of each subprogram, a cross reference map, and any error diagnostics. A new language processor should be able to perform the same functions. The error diagnostics must be complete, informative, and correct.

If a preprocessor is used, then an intermediate file of generated Fortran code will be produced. It is often difficult or impossible to map errors detected by the Fortran compiler back to the original source code. While such difficulties may be impossible to eliminate, they should be minimized as much as possible.

Other support tools many be needed in addition. For example, structured programs require indentation in order to achieve full readability. Extensive revision of programs that were originally nicely indented and laid out can result in a tangled mess. The availability of an automatic formatting program which will reformat a program to follow a well defined set of indentation and layout rules can be a big factor in gaining acceptance for a new programming language. Programmers can be lazy and if the new language will take care of program formatting automatically, then that will often be a big advantage over Fortran. Formatters also can produce code that is uniform across an entire project involving many programmers. However, such formatters should handle comments reasonably and allow the user to override the formatter defaults when necessary.

Many current operating systems provide powerful interactive debugging facilities. Often, these debugging facilities interface only with assembly language or Fortran. A new language should either provide its own debugging facilities, or interface easily with the current debugger. If the new language provides its own enhanced debugging tools, as well as an interface to the existing debugging facilities, so much the better.

5. It should run efficiently and generate efficient code.

Production Fortran compilers usually run fast and almost always produce efficient object code. While execution speed is no longer as critical as it once was, it is still important. If the code produced by the new language processor is not as efficient as that produced by Fortran, then there is a reason to reject the new language. Perhaps just as important as the execution speed is the compilation speed. Programmers do not like to wait for their programs to be compiled. If a Fortran preprocessor is used, the total compilation must necessarily be longer than the Fortran compilation time alone. If the preprocessor does much more than double the total time, it is likely to be rejected.

6. It should be as flaw free as possible.

Unwilling users can go to great lengths to find something wrong or inconvenient with the use of a new language or tool. If acceptance is to be gained, legitimate complaints about the operation of a new tool cannot be dismissed lightly. If the new product is to be commercially purchased, it should first be thoroughly evaluated to be sure there will not be grounds for serious complaints. If the source code for the new product is available, there is more flexibility since the complaint can be handled by fixing the problem. There should be compromise, however. While no complaint can be dismissed without at least considering it, unjustified complaints should not be allowed to result in either the rejection of a product or unnecessary revision in its standard operation. The result of this constructive give and take process will not only increase the chances for acceptance, but will also likely lead to a much better end product.

7. The benefits should justify leaving the current system.

The introduction of a new language or tool will require an initial learning curve for users not already familiar with it. The shorter the learning curve, the greater the chance for acceptance. If the new language is implemented as a preprocessor, for example, then there will also be some overhead in compilation time and ease of use (relating generated Fortran back to the original code, for example). In addition, there will likely be a great deal of inertia to overcome. The benefits produced by switching to the new language or tool should be great enough to outweigh all of these costs.

If a great enough benefit can be obtained from a new tool or language, then it is still possible for it to gain acceptance even though it fails one of the previous criteria. However, if the benefits are only marginal (or perceived to be marginal), then failing any of the criteria will likely lead to rejection.

Gaining acceptance for Ratfor

The above guidelines were discovered (or rediscovered) in the process of gaining acceptance for Ratfor. While in retrospect the rules may seem obvious, had they all been known and believed beforehand, that process would have been much easier. Ratfor already complied with some of the guidelines, but failed to meet others until modifications were made. The following paragraphs describe how the new version of Ratfor complies with these guidelines.

Probably the main thing Ratfor has to offer is that it is designed to work on top of a Fortran environment. The step from Fortran to Ratfor is much easier to take than the step from Fortran to Pascal. The basic data types of Ratfor are the same and the control structures are typical of any structured programming language (e.g., if then else, while, repeat). Thus, when learning Ratfor, the Fortran programmer is confronted only with the new control structures, and does not have to face a baffling array of new, complicated data types. In fact, most Fortran programmers can start to learn Ratfor by converting a few old programs from Fortran into Ratfor.

In addition, since Ratfor is translated directly to Fortran, it will interface directly with anything that Fortran will. Thus, any existing system services, libraries, or utility packages that Fortran can use are equally available to Ratfor. This eliminates one argument that can often be used against other languages such as Pascal.

On the other hand, the distributed portable version of the Ratfor preprocessor was definitely not easy to use. There was a two-step process the programmer has to explicitly invoke when compiling a program: Ratfor to Fortran, then Fortran to object code. The Ratfor to Fortran step usually could not use the same system conventions as the Fortran to object code step. It was also very difficult to debug Ratfor programs translated by the distributed version of the preprocessor. The error diagnostics, while informative, were very difficult to relate back to the offending line of the original Ratfor code.

Fortunately, the Ratfor preprocessor itself was written in Ratfor, and was thus easily modified to overcome its shortcomings. The error diagnostics were changed so that they would print the message, the offending line, and an exact line number within a routine on the user's interactive terminal. Several new preprocessor options were also implemented in the form of user specified switches. These switches allow the programmer to include Ratfor source line numbers and comments in the generated Fortran code, as well as to include statements for debugging that are compiled only when the corresponding switch has been enabled. Finally, the new version of the preprocessor has been installed on the operating system so that it can be invoked in the same manner as the Fortran compiler or any other system utility. By using the appropriate command, the Ratfor user can now either invoke the Ratfor to Fortran step only or the entire Ratfor to object code translation in a single command to the operating system.

Modifying the preprocessor itself so that it is in an easy to use form was only the first step in gaining acceptance for Ratfor. While the original Ratfor paper gave a fairly complete description of an early version of the language [2], the current version of Ratfor has a number of extensions over the original. A much more complete Ratfor document (as well as source code for the preprocessor) had been obtained from the Advanced Systems Research Group at the Lawrence Berkeley Laboratory [3]. That document was extensively rewritten and reorganized to present Ratfor in a more orderly fashion, as well as to describe the new features. A fairly large library of useful Ratfor routines was also supplied by the Lawrence Berkeley group. This library was expanded (and currently includes nearly 40 routines) and a description of each routine was included in the user's document. The resulting document, almost 75 pages long, now serves as a complete and definitive Ratfor programmer's manual.

Ratfor is easiest to read if it is well laid out and follows a consistent set of indentation rules. The other project members requested that an automatic formatting program be obtained or written. Since no other Ratfor formatter was known to exist, one was written one using the original version of the Ratfor preprocessor as a starting shell. Using this shell, he Ratfor formatter was written in less than a week. The formatter is not a trivial program and it is extremely doubtful that it could have been written so quickly had the the language been Fortran instead of Ratfor.

The final tool added to the Ratfor arsenal was a cross reference generator. A cross reference generator is useful for detecting typographical errors (e.g., 'lisths' instead of 'listhd'), as well as aiding the understanding of the overall variable usage of someone else's program. Again starting with the Ratfor preprocessor as a snell, the cross reference generator was written in less than two days. The fact that it was possible to write both the formatter and the cross reference generator in such a short time, as well as the ease of maintaining the original Ratfor preprocessor code, can be attributed largely to the language benefits of Ratfor.

The efficiency of both the preprocessor and the resulting generated code was also of concern. The original distributed version of Ratfor took 175 seconds to translate itself to Fortran on a Digital Equipment VAX/VMS system, while the Fortran compiler took only 81 seconds to produce the object code from the resulting generated code. Using some suggestions from Software Tools, as well as from Comer [4], the preprocessor was modified so that the final production version takes only 36 seconds to translate itself to Fortran. Thus, the total translation time for Ratfor to object code is only 44% greater than Fortran to object code time alone.

Earlier studies of the efficiency of code generated by Fortran preprocessors had indicated a fairly low overhead cost [5] in both speed and space with Ratfor. In an effort to remove what little overhead there was, the Ratfor preprocessor was modified to generate the "IF - THEN - ELSE - END IF" Fortran-77 construct recognized by the VAX Fortran compiler. Results of benchmarks showed virtually no difference in execution speed of Ratfor code and carefully hand coded equivalent Fortran.

Conclusions

An entire Ratfor environment has been created within the normal usage conventions of the operating system to comply with the acceptance guidelines. The enhanced version of Ratfor is much easier to use and is better documented than the original version of Ratfor. In addition to the obvious benefits of improved control structures and free form input, the maintainability of Ratfor has also been shown by the ease of the modifications to the preprocessor itself, as well as the fast implementations of the formatter and cross reference generator.

While this paper describes the specific experiences of getting a group of Fortran programmers to accept Ratfor, the results should apply to any new software tool in any environment. Any new tool must work well within the conventions of the existing system and provide significant benefits before it stands a chance of being accepted. By recognizing this beforehand and taking appropriate steps, it should be possible to dangle a big enough carrot to get even the most reluctant user to try a useful new software tool. And once Fortran programmers have accepted a tool such as Ratfor, the next step to Pascal or other language tool will not seem as big.

References

[1] B. Kernighan, and P. Plauger, Software Tools, Addison Wesley Publishing Co., Reading, MA, (1976).

[2] B. Kernighan, "Ratfor - a Preprocessor for Rational Fortran", Software - Practice and Experience, Vol. 5, 395-406 (1975).

[3] The Advanced Systems Research Group, Computer Science and Applied Mathematics Department, Lawrence Berkeley Laboratory, Berkeley, CA., (1979).

[4] D. Comer, "MOUSE4: An Improved Implementation of the Ratfor Preprocessor", Software - Practice and Experience, Vol. 8, 35-40 (1978).

[5] R. Meeson, and A. Pyster, "Overhead in Fortran Preprocessors", Software - Practice and Experience, Vol. 9, 987-999, (1979).