A FORTRAN IV To QuickBASIC Translator

Rizaldo B. Caringal and Phan Minh Dung
Division of Computer Science
Asian Institute of Technology
G.P.O. Box 2754
Bangkok 10501, THAILAND

Abstract

This paper describes the design and implementation of an automatic translator from standard FORTRAN IV to QuickBASIC, a structured form of the programming language BASIC. The translator makes two passes on the input program before finally generating the translated program. The converter not only performs lexical, syntactic and limited forms of semantic analyses on the source program, but it also recovers from any errors encountered. It was implemented using the C programming language in the Disk Operating System (DOS) environment and was successfully ported to UNIX. Furthermore, the contingencies to cope with other dialects of the source language have been defined, and the problems encountered in the implementation process are discussed.

Keywords: Automatic Translator, Converter, FORTRAN, QuickBASIC, Portable

Introduction

The advent of FORTRAN IV as the standard programming language for writing numerical and engineering applications produced numerous libraries of subprograms of the said language [14]. However, since FORTRAN IV emerged during the period when microcomputers were not even heard of, these libraries are mostly stored and utilized in mainframes. The unavailability of mainframe computing facilities, not to mention the high cost of computing if they are at all available, restrict the access of these vast software libraries. Additionally, the migration of most programmers to structured programming languages made FORTRAN IV libraries unusable in several cases. A viable solution therefore is to make these FORTRAN programs available in cheaper, more accessible hardware, e.g. microcomputers.

There can be two possible solutions to this situation. One is to provide a mixed programming environment, where a host language, e.g., Pascal, QuickBASIC etc., can access subroutines written in FORTRAN IV. The main constraint in this approach is that languages to be mixed should operate on the same environment. This still requires the use of mainframe computers [3].

An alternative solution would be to translate FORTRAN programs into another high level programming language either

automatically or by hand. According to Freak [5], translating programs by hand poses the following disadvantages: the logical structure of the original program might not be preserved; there is a risk of introducing bugs to the program; problems regarding documentation may arise; the process of translation by hand is a tedious one; and the reliability of the resulting program is unassured. On the other hand, Slape and Wallis [10], enumerated several problems associated with translating programs automatically. First, the translation process is technically infeasible if the languages being converted are too radically different; second, translation is unattractive if majority of the users are likely to rewrite most of their programs to enhance specifications; and third, the generated programs are not sufficiently idiomatic in their use of the target language to make subsequent maintenance of the target language version an attractive proposition.

The choice of QuickBASIC, which is gaining vast popularity in this part of the world, as the target language alleviates the first difficulty. However, there are still some incompatibilities between FORTRAN and the said language. The last two problems of translating programs automatically are not expected to be encountered much since the translations will be made between standard software libraries. This makes automatic program translation an attractive proposition in addition to offsetting the negative factors mentioned by Freak [5].

The features of the source language were limited to the ANSI X3.9-1966 FORTRAN definition [11,12,13] - also known as FORTRAN 66, while the target language was based on the Microsoft QuickBASIC version 4.5 [8]. An attempt was made to map every language construct of FORTRAN to QuickBASIC, although some problems were encountered and compromises had to be resorted to. Specifically, FORTRAN's input and output facilities, and EQUIVALENCE statement were implemented differently in the corresponding QuickBASIC target code. Passing of subprogram names as parameters to other subprograms was not supported.

The structure of FORTRAN and the need to make the translator simple led to a two pass translation scheme. Nevertheless, the translation process itself is fast enough to be comparable with some existing compilers. The implementation language C allowed the translator to be portable to machines supporting the standard definition. The translation process is automatic in the sense that only the source program and the device parameters' file are needed to produce the target program. A runtime library is provided for QuickBASIC programs, thus fully supporting FORTRAN's intrinsic and basic external functions. Complete lexical and syntactical analyses, in addition to limited semantic checking are performed by the system with full error recovery. In addition, a user interface is provided for the DOS environment, facilitating the translation of FORTRAN source codes, the compilation of the resulting QuickBASIC programs and the linkage with the standard and runtime libraries.

Differences Between FORTRAN and QuickBASIC

QuickBASIC is an extended definition of the programming language BASIC providing features such as subprograms and functions, user-defined data type, recursion, and flexible array dimensioning, much more advanced than the original. Although the traditional constructs of the language have been retained, additional control statements have been included to support structured programming [8].

Structurally, FORTRAN and QuickBASIC programs share common features. Both languages require a main program optionally followed by subprograms - either subroutines or functions. The major differences between the two lie with other aspects namely: control structures, data structures, name structures and syntactic structures.

Although the control statements of the two languages are syntactically different, only two statements of FORTRAN are not supported in QuickBASIC - the Arithmetic IF and the Assigned GOTO. Among the data structures supported by FORTRAN IV, the complex and Hollerith data types have no counterparts in QuickBASIC. Additionally, the data type integer in FORTRAN can be assigned another type - label, through the Assigned GOTO statement. In both languages, implicit declarations of variables are allowed. FORTRAN permits implicit integer types of variables starting with letters I, J, K, L, M, and N, while the rest are assumed to be of type real. In contrast, QuickBASIC only allows implicit types for variables of type real.

Sharing of memory locations by several variables in the same program block is achieved in FORTRAN through the EQUIVALENCE statement. When two variables are declared to be equivalent, they will be stored in the same memory space if they have the same size or share some common locations if their sizes differ. This feature is not supported in QuickBASIC.

Focusing on the syntactic structures, FORTRAN has two main weaknesses [7]. First, it ignores blanks throughout the program, except in Hollerith constants, and second, it has no concept of reserved words. In QuickBASIC, blanks are considered as delimiters of lexemes while keywords are treated as reserved. This implies that keywords in this language are forbidden to be used as program identifiers.

Lastly, there are differences in the input and output formats, and operations of the two languages. In FORTRAN, I/O is record-oriented, that is, each I/O statement accesses a new record on the external medium with or without format specifications. In QuickBASIC, I/O can either be stream-oriented or record-oriented.

Translation Rules of the System

A set of translation rules mapping all the considered language constructs of FORTRAN into equivalent QuickBASIC structures was defined [2]. The rules were grouped according to some general categories, such as: program form, data types, data and procedure identification, expressions, statements, and procedures and subprograms, describing the language features of the source language and the corresponding structure of the target language. For each feature of the language, the following information are given: the facilities available in both languages, the translation rule, and an optional example to clarify more complicated rules.

There are certain rules which were difficult to implement. One is the support for complex data types where a three-fold problem is implied - representation of constants, variables and function references. This problem was resolved by defining a new data type in BASIC called COMPLEX which is actually a record with two single precision fields - one is REAL for the real part and another is IMAG for the imaginary part. The following type declaration is included in every BASIC program which uses complex data types:

```
TYPE COMPLEX
     REAL AS SINGLE
     IMAG AS SINGLE
END TYPE
```

Assignment of values to such variables will also entail two operations - one for the real part and another for the imaginary part. Constants are handled depending on where they are used in the program, usually in expressions, and data initialization statements. Given the complex variables A and B, the following FORTRAN statements

```
DATA A / (1.0, 2.0) /
B = (3.0, 4.0)
```

will be translated to QuickBASIC as:

```
READ A.REAL, A.IMAG
DATA 1.0, 2.0
VAR0001.REAL = 3.0
VAR0001.IMAG = 4.0
B = VAR0001
```

where VAR0001 is a temporary complex typed variable created by the system. Note that the assignment of values in BASIC are always done in the field level of the record since a record itself cannot be assigned a value directly. Similarly, FORTRAN's DATA statement with a complex type argument was translated in such a manner that the real and imaginary fields of the record are initialized separately.

For a multi-operator complex expression, the operations have to be broken down such that one is evaluated at a time, and the intermediate results are stored in temporary variables. Assuming that A, B, and C are complex variables, the following FORTRAN statement

```
A = A + B * C
```

will be translated to QuickBASIC as

```
VAR0001.REAL = (B.REAL * C.REAL) - (B.IMAG * C.IMAG)
VAR0001.IMAG = (B.IMAG * C.REAL) + (B.REAL * C.IMAG)
VAR0002.REAL = A.REAL + VAR0001.REAL
VAR0002.IMAG = A.IMAG + VAR0001.IMAG
A = VAR0002
```

where VAR0001 and VAR0002 are temporary variables of type complex created by the translator. In addition, if a real typed component is involved in a complex expression, then the said component is treated as a complex type with its value as the real part and zero as the imaginary part.

Since a user defined type cannot be returned by QuickBASIC functions, a FORTRAN complex typed function is translated by using a subroutine and by adding an extra variable of type complex to the list of the subroutine's parameters. This extra variable will contain the value that is supposed to be returned by the function. The subroutine is invoked prior to the original statement where the function was referenced and consequently, the extra variable replaces the occurrence of the function reference. A more complicated case of this would be a statement involving an expression with nested function references, as given in the following example, assuming X and CEXP(X) are of type complex.

```
X = CEXP(CEXP(CEXP(X)))
```

One solution can be to evaluate the innermost function reference first and store the result in a temporary variable, and use this temporary variable to evaluate the second innermost reference, and so on. The corresponding QuickBASIC statements are:

```
CALL CEXP(X,VAR0001)
CALL CEXP(VAR0001,VAR0002)
CALL CEXP(VAR0002,VAR0003)
X = VAR0003
```

A similar case was encountered in evaluating nested FORTRAN minimum and maximum family of functions. Since such FORTRAN functions allow variable numbers of parameters, corresponding translations in QuickBASIC use single dimensioned arrays to contain the original parameters. The arrays together with their cardinalities are the ones passed to the functions. The reason for this convention is to cope with the limitation of QuickBASIC functions which do not allow a variable number of parameters to be passed.

A FORTRAN's input/output list has its own inherent difficulty in translation. First, there is the DO-implied list which uses an implied loop to address some or all elements of an array. Second, an array name can be specified in a list by itself, whereby all the elements are accessed in a column major order strategy. Both of these cases were handled using QuickBASIC's FOR..NEXT statement. For the FORTRAN statements

```
DIMENSION A(10), B(10,10)
WRITE (6) ((B(I,J), A, I=1, 10), A, J=1, 10)
```

the corresponding QuickBASIC code will be:

```
DIM SHARED A(10)
DIM SHARED B(10,10)
DIM SHARED VAR0001 AS INTEGER

FOR J = 1 TO 10
    FOR I = 1 TO 10
        WRITE #6, B(I,J)
        FOR VAR0001 = 1 TO 10
                WRITE #6, A(VAR0001)
        NEXT VAR0001
    NEXT I
    FOR VAR0001 = 1 TO 10
        WRITE #6, A(VAR0001)
    NEXT VAR0001
NEXT J
```

where VAR0001 is an integer variable created by the system.

Compromises were made in mapping some constructs of FORTRAN to QuickBASIC. First, only unformatted I/O constructs of FORTRAN are translated to QuickBASIC. This implies that formatted I/O constructs of the source program are first converted to unformatted ones, and then translated to the target language. This led to the ignorance of the occurrences of FORMAT statements in FORTRAN programs. Second, difficulty was encountered in translating FORTRAN's EQUIVALENCE statement. Direct sharing of memory locations in the same QuickBASIC module is not possible. As a compromise, only scalar variables of the same type were allowed to be parameters of this statement. Variables of different types are ignored and warnings are issued. The technique utilized was to assign the new value acquired by one of the variables in the equivalence class to all the other variables in the same class. Lastly, the mechanism of FORTRAN to pass subprogram names as actual parameters to subprograms is unsupported by the translator. Problems were encountered in the effort to translate this construct of FORTRAN to QuickBASIC since code addresses cannot be passed as parameters to subprograms in the latter.

The Translation System

The translation system consists of two major modules: the first pass which transforms the FORTRAN source code into an intermediate form describable using the Backus-Naur Form (BNF) notation; and the second pass which converts the intermediate form into the QuickBASIC target code.

The First Pass

This module facilitates the uniformity of the scanning process for the different tokens of FORTRAN. A number of irregularities of the FORTRAN definition led to the design of this module. First, blanks are not significant in FORTRAN programs except in Hollerith constants, hence there may or may not have boundaries between tokens. Another implication of this is that a single token might have embedded blanks within itself. These things complicate the lexical analysis process. Below are three equivalent statements in FORTRAN but are presented differently.

```
INTEGERA,B,C,D,E                          (1)
I N T E G E R A , B , C , D , E           (2)
INTEGER A, B, C, D, E                     (3)
```

We might have written the statement as in (3), but (1) and (2) have the same meaning. One task of this module is to convert every form of a statement into a standard one by deleting all unnecessary blanks, and correspondingly, unnecessary white spaces such as tabs, control characters and the like.

Another feature of a FORTRAN statement is that it uses the 6th column of a line to indicate whether or not the current is a continuation of the previous line. To achieve uniformity, this module merges multiple line statements into one. It uses a special character - %, which is not part of the FORTRAN IV character set, to delimit lines in multiple line statements. Statement lines (4) and (5) below are converted into (6).

```
100    INDEX = INDEX * 2 + INDEX * 3 +          (4)
     *         INDEX * 4                       (5)
100INDEX=INDEX*2+INDEX*3+%INDEX*4              (6)
```

Other features of this module are: it deletes comment statements; it marks the end of each FORTRAN statement; and it ignores statement labels appearing in continuation lines.

The Second Pass

The second pass of the translation system is responsible for converting the intermediate form created by the first pass into an equivalent QuickBASIC target code. It is further divided into the following submodules: the lexical analyzer which recognizes tokens of FORTRAN from the intermediate form and passes them to the syntax analyzer; the symbol table manager which initializes

and maintains the necessary fixed and variable symbol tables; the syntax analyzer or the parser which checks for syntax correctness by parsing the FORTRAN source code based on the BNF description; and the code generator which produces the desired QuickBASIC target code. All errors which might occur in the translation process are reported by the error handling and recovery routine.

Lexical Analysis

The task of a lexical analyzer or a scanner is to recognize tokens of a language. If such tokens have attributes, i.e., values, then they should also be returned by the module. Due to certain irregular features of FORTRAN, a scanner cannot be designed in a straightforward manner [4]. Token boundaries in FORTRAN are not determined by uniform rules. But due to the transformation done in the first pass, no more boundaries exist between tokens and they must now be recognized depending on the context in which they are used. Consider the example below:

```
ASSIGN100TOTAL                                  (7)
ASSIGN  100   TO   TAL                          (8)
```

Statement (7) has four tokens as shown in (8), two keywords (ASSIGN and TO), one label (100), and one identifier (TAL).

Another difficulty with FORTRAN is that keywords are not reserved. This means that keywords can also be used as identifiers, e.g., variable names, procedure names, etc. Consider the example below.

```
DO200I=1,100                                    (9)
DO  200  I  =  1  ,  100                        (10)
DO200I=1.100                                    (11)
DO200I  =  1.100                                (12)
```

Statement (9) has seven tokens as shown in (10), one keyword (DO), one label (200), one identifier (I), two special characters ("=" and ","), and two numbers (1 and 100). But statement (11) has only three tokens, one identifier (DO200I), one special character ("="), and one number (1.100). The keyword DO in (9) is not recognized until the character "," is encountered. Similarly, DO200I is recognized as an identifier when the character "." is scanned.

An interesting feature of FORTRAN is that aside from the line number, a statement usually begins with a keyword. The only exception is the assignment statement which begins with an identifier. The general approach with the scanner is to recognize a keyword at the beginning of a statement. If a keyword is identified, then the rest of the statement is scanned for an "=". If it is found, then the statement is assumed to be an assignment statement, otherwise the keyword is accepted. If an identifier is recognized in the first place, then an assignment statement is assumed. This is, however, not true for all cases as illustrated by (9). For the DO keyword to be

identified, a "," is scanned instead of an "=".

After recognizing the first token of a statement other than the statement label, a keyword can no longer occur within the statement except in two cases: in an ASSIGN statement where TO should follow after a label; and in a typed function statement where the keyword FUNCTION should follow after any of the type statements COMPLEX, DOUBLE PRECISION, INTEGER, LOGICAL or REAL. These cases are handled accordingly.

Since FORTRAN is not context free, it is necessary to distinguish the different usages of identical symbols. One case is the distinction between an assignment statement or a statement function definition, e.g.:

```
IF(I) = 2                                    (13)
```

from the arithmetic and the logical IF statements:

```
IF(I) 1,2,3                                  (14)
IF(I) GOTO I                                 (15)
```

Note that in (13), IF is recognized as an assignment statement if it was declared to be an array previously, otherwise it is recognized as a statement function definition. Additionally, an identifier followed by a "(" is recognized as an array name if it is in a DIMENSION, COMMON or TYPE statement, otherwise it is recognized as a function name.

IF in (14) is recognized as an arithmetic IF statement since the character following the rightmost close parenthesis of the logical expression is a digit. Similarly, (15) is recognized to be a logical IF because the character following the rightmost close parenthesis of the expression is not a digit. The rules for recognizing a new statement has to be applied for identifying the tokens in a logical IF statement since a new statement begins after the IF logical expression.

Numbers and labels are also distinguished from each other since they can be of the same form, that is, a series of digits. A series of digits is recognized as a label in the following cases: if it is the first token of a statement, if it is in any GOTO statement, if it supersedes a DO keyword, if it comes after the rightmost close parenthesis of the arithmetic expression in an arithmetic IF statement, and if it is the second argument of a READ or WRITE statement. Otherwise, a series of digits is recognized to be a number.

A finite state automaton was constructed to efficiently distinguish the different tokens of FORTRAN [2].

Symbol Tables and Symbol Table Management

Two fixed and two variable symbol tables are needed for the translator. The two fixed tables are used for storing QuickBASIC reserved words, and FORTRAN intrinsic and basic external function names. On the other hand, one variable symbol table is needed to keep track of all the identifiers used in the FORTRAN program, i.e., variable names, array names, procedure names, function names, block names and labels; and the other is needed to keep track of open files or devices in the program.

As much as possible, the translator preserves the identifiers of the source program in the equivalent target code. However, keywords in QuickBASIC are reserved. Thus, QuickBASIC keywords utilized as identifiers in a FORTRAN program cannot be copied directly to the corresponding QuickBASIC target program. A name generator was constructed to produce a unique identifier name to be used instead of the reserved word. The said generator is also utilized by the code generator to create temporary variable names needed in some translation processes.

Since the translated program is supposed to operate in a microcomputer, it is imperative to minimize the number of identifiers in the QuickBASIC program due to limited memory space. To implement this, the concept of pooling all temporary variables, depending on the type and structure, i.e. whether scalar or array, has been employed. One list is maintained for each of the classification. If a temporary identifier is needed, then the appropriate list is checked. If one is available, then it is returned and deleted from the list, otherwise the name generator is invoked to produce a new name. After a set of temporary variables have been used in translating a statement or a group of statements, they are freed and returned to the lists.

The table for FORTRAN built-in functions contains information for checking the correctness of said functions, i.e., the name of the FORTRAN function, the types of arguments, the number of arguments and the type of the function. In addition, it keeps track of the corresponding function names in QuickBASIC, whether built-in or externally supported, which are needed during the translation process.

One of the variable symbol tables is used to keep track of all identifiers used in the FORTRAN program. It contains information such as an identifier's name, type, etc., and various parameters depending on the kind of the identifier, e.g., in an array, the number of dimensions and the bounds for each dimension. The operations needed for this table are to search for and insert a given name. This table is always consulted whenever an identifier is encountered in the translation process. The other variable symbol table is used to record the current open devices or files in the FORTRAN source program. These are predefined correspondences between device and file numbers, and actual devices and filenames.

Syntax Analysis and Code Generation

This module is the largest, approximately 60% of the whole translation system, and the most complicated. It parses and checks each of the FORTRAN statements in the source program according to the defined BNF rules. If an error is found, the error handling and recovery module is invoked. After recovering from the error, the control is passed back to the syntax analyzer so that succeeding statements can be processed.

The strategy employed in translating a FORTRAN statement to QuickBASIC is to first recognize and parse the statement of the former and check, according to the defined rules, if it can be translated directly to the latter. If this is feasible, then the equivalent QuickBASIC statement is constructed immediately and written to the output file, otherwise, some temporary file is created where intermediate results are stored. When a FORTRAN statement is encountered which will complete the translation process, the generated code stored in the temporary file is appended to the output file. In this sense, the code generation module is tied up with syntax analysis. Additionally, there are some language constructs where transformations have to be done before actual translation can be performed. It should not be surprising that there are some FORTRAN statements whose equivalent QuickBASIC code exceeds twenty statements each.

A good example of postponing code generation until the logical end of the main program or a subprogram is the declaration of variables. This is due to two factors. First, since keywords in QuickBASIC are reserved, new identifiers have to be created for each of the QuickBASIC keywords appearing in the FORTRAN source. If the default types of the QuickBASIC keyword (with respect to the FORTRAN program) and the new identifier are not the same, then the latter should be explicitly declared in the QuickBASIC program. Second, there are a number of circumstances where new variables are created in the process of translating certain statements of FORTRAN. These new variables are presumed to be of specific types, hence explicit declaration is necessary.

To simplify the implementation of the FORTRAN BNF definition, one logical function was created for each of the BNF rules. These functions return true if the parsing procedure was successful, otherwise they return false. Two options are provided for translating FORTRAN programs - one for specifying a main program optionally followed by subprograms, and another for subprograms only. This resulted in two starting BNF rules depending on the option chosen.

Error Handling and Recovery

The tasks of this module are to communicate appropriate messages regarding the errors encountered during the translation process and to ignore the statements where the disorders occurred. The messages that can be relayed by the translator are

classified into two main groups: system errors, and translation errors and warnings. The first group of errors are considered severe and causes the system to cease execution. These usually involve hardware dependent factors such as memory space and disk space. The second group of errors are those arising from the lexical, syntactic and semantic analyses performed by the system on the source program. Warning messages are given for those language constructs of the FORTRAN program which are either supported differently or not translated by the system, while error messages are for actual violations of the FORTRAN definition.

User Interface

An optional user interface module is provided for the Disk Operating System. The purpose of this is to facilitate file handling and maintenance, editing, the translation process itself, the specification of options, and the compilation of the QuickBASIC programs together the linkage of appropriate libraries.

Conclusions

The size of the translation system is about 210 K in the DOS environment. This leaves approximately 400 K for temporary storage and processing. The translation speed is acceptable - for a 5000 line FORTRAN program without comments, it averaged 41 lines per second on a microcomputer with an Intel 80286 processor. The first pass of the translation takes approximately 16% of the total processing time. It is expected that the performance of the translator will improve if it will be converted to a one pass system. On the average, the resulting QuickBASIC programs are 76% larger than the FORTRAN programs.

Several FORTRAN routines, from numerical algorithms to engineering applications, have already been converted without modifications to QuickBASIC using the translator system. The resulting QuickBASIC programs have been compiled directly and linked with the necessary libraries. The results of executing these programs confirmed the correctness of the translation process.

The technology of automatic high level programming language translators had emerged in software engineering. This work, together with other translators such as - Pascal To C [1], FORTRAN To Pascal [5], Small Euclid To Pascal [9], and Sail To C [6], are some of the indications that the automatic translation of one high level programming language to another is a viable solution to the problem of program conversion.

The contingencies to handle FORTRAN 77 constructs have already been defined in [2]. The structures of the latter FORTRAN definition together with the structuring of source languages's control statements are planned to be incorporated in the next version of the system.

References

1. BOTHE, K., B. HOHBERG, CH. HORN, and O. WIKARSKI (1989), A Portable High-Speed Pascal To C Translator, **Sigplan Notices,** Vol. 24, No. 9, pp. 60-65.

2. CARINGAL, R. B. (1990), **A High Level Programming Language Translator From FORTRAN IV To QuickBASIC**, Masters Thesis, Asian Institute of Technology, Bangkok, Thailand.

3. EINARSSON, B. and W.M. GENTLEMAN (1984), Mixed Language Programming, **Software - Practices and Experience,** Vol. 14(4), pp. 383-395.

4. FELDMAN, S.I. (1979), Implementation of a Portable Fortran 77 Compiler Using Modern Tools, **ACM SIGPLAN Notices,** Vol. 14, No. 8, pp. 98-106.

5. FREAK, R.A. (1981), A Fortran to Pascal Translator, **Software - Practices and Experience,** Vol. 11, pp. 717-732.

6. LEMKIN. P.F. (1987), **PSAIL - A Portable SAIL to C Language Compiler,** Image Processing Section Laboratory of Mathematical Biology, DCBDC National Cancer Institute.

7. MacLENNAN, B.J. (1983), **Principles of Programming Languages,** Holt, Rinehart and Winston.

8. MICROSOFT. (1988), **BASIC Language Reference,** Microsoft Corporation.

9. PINTELAS, P.E., K.P. VENTOURIS and M.D. PAPASSIMAKOPOULOU (1989), A Translator from Small Euclid To Pascal, **Sigplan Notices,** Vol. 24, No. 5, pp. 93-101.

10. SLAPE, J.K. and P.J.L. WALLIS (1983), Conversion of Fortran to ADA using an Intermediate Tree Representation, **The Computer Journal,** Vol. 26, No. 4, pp. 344-353.

11. USASI Sectional Committee X3. (1964), **FORTRAN vs. Basic FORTRAN,** Communications of the ACM, Vol. 7, No. 10, pp. 591-625.

12. USASI Sectional Committee X3. (1969), **Clarification of FORTRAN Standards - Initial Progress,** Communications of the ACM, Vol. 12, No. 5, pp. 289-294.

13. USASI Sectional Committee X3. (1971), **Clarification of FORTRAN Standards - Second Report,** Communications of the ACM, Vol. 14, No. 10, pp. 628-642.

14. VAN DER LAAN, C.G. (1982), Programming In Algol 68 (as a host) and the Usage of FORTRAN (program libraries), **The Relationship Between Numerical Computation and Programming Languages,** North-Holland Publishing Company, IFIP.