Design of a FORTRAN to C Translator

Fred Goodman
Great Migrations LLC

Abstract

Waite and Goos in Compiler Construction [1] define a compilation as "a sequence of transformations (SL,L1),(L1,l2),...(Lk,TL) where SL is the source language and TL is the target language." This article describes and motivates the design of PROMULA.FORTRAN, a compiler whose source language is FORTRAN, whose intermediate language is a reverse polish pseudo-code, and whose target language is C. The major objectives of the design are: (1) that the C output produce the same results as the FORTRAN original, (2) that the user be able to modify the FORTRAN dialect description, and (3) that the user be able to modify the look of the C output. The bulk of the paper deals with objective (3). A surface-form description language is presented along with an example translation. Given that objective (1) above must always be met, there are three possible views of the C output: that it be as efficient as possible to compile, that it be as "C-like" as possible, and that it be as much like the original FORTRAN as possible. These views are discussed along with their impact on the design of the runtime library. Finally, surface-form representations are given for each in terms of the example translation.

This paper was first published in The Journal of C Language Translation; Volume 1, Numbers 3 and 4; December, 1989 and March, 1990.)

by Fred Goodman, Promula Development Corporation

Introduction

In the November/December 1987 Micro/Systems Journal, A.G.W. Cameron reviews a FORTRAN to C translator. In that review, he takes the position that the only difference between a translator and a compiler should be that the compiler translates the source code into assembly code while the translator takes it to a higher level language. This article discusses a FORTRAN to C translator, PROMULA.FORTRAN, which takes precisely this approach to translation. First, it compiles the FORTRAN source code into a low level pseudo-code, much like the pseudo-code produced by the first pass of contemporary compilers. Second, it optimizes that code again using the same techniques as would be used by a conventional compiler. Third, it does code generation; but the code generated is not machine code, it is C. What Cameron does not discuss, however, is that though the processes of translation and compilation are largely the same, the objectives of users of translators are much more varied than those of compiler users. The user of a compiler does not care what the output code looks like—it is merely an intermediate step. Though some users of PROMULA.FORTRAN also view the output C merely as an intermediate step, most view it as a final step. They care very much what the C looks like, because they intend to discard the FORTRAN original once the translation is completed. In general, users of PROMULA.FORTRAN can be divided into three broad classes:

1. Those who want to continue using their present FORTRAN dialect as their programming language. For these users the C output is of no importance as such. It should be designed to compile as quickly as possible.
2. Those who are presently FORTRAN programmers, but who want to become C programmers. For them the C output should be as close to the original FORTRAN as possible to ease the transition.
3. Those who are C programmers who must now take over a FORTRAN code. For them the C output should look as much as a standard C program as possible.

The techniques used to produce the target language are based on the work done in the area of re- creation of source code. In particular, P. J. Brown [2] discusses the re-creation of source code from reverse polish notation. This paper reviews that work and shows how it can be easily extended via a "surface-form description language" to allow not only for the surface form variation needed to accommodate different user biases, but also to give the final user of the translator the ability to tailor the translation to his own needs. To motivate the discussion of the surface form description language, a brief discussion of the history and design objectives of PROMULA.FORTRAN is given.
The discussion is presented in journalistic style. The purpose of the presentation is to describe how a translation is achieved with emphasis on the output side. This emphasis is chosen because most readers of this article will already be familiar with computer language input processing. Each section becomes increasingly detailed and thus perhaps more tedious to follow; however, the basic ideas are presented early for those who only want an overview of the approach.

Background

PROMULA.FORTRAN has been under development since 1982. It was a child of necessity. The original FORTRAN compilers available for microcomputers were pathetic by comparison to their mainframe ancestors. This was not true of the early C compilers, especially after the "large memory models" became available. We are consultants, and were often asked to migrate mainframe FORTRAN codes to the PC. This migration task proved much easier to achieve via translation to C. Given this need to actually use the translator as a substitute for the existing FORTRAN compilers, our first design goal was that the translated version had to produce the same results as the original FORTRAN version. Our original view of the C output was that it was merely an intermediate step. It had to be efficient and readable, but not necessarily maintainable.
The initial version of PROMULA.FORTRAN took the approach of examining each FORTRAN statement in the source program and then producing from it an equivalent statement, or statements, in C. Our view of translation was that it was basically a string manipulation problem—it was nothing like compilation. Given constructs like implied do loops, assigned GOTO statements, and the hiding of characters in FORTRAN 66, it became clear that if any degree of completeness were to be achieved, then the program units had to be processed as a whole and then examined. Statement by statement processing simply did not have legs. The resultant C codes either would not compile at all, or did not give the correct results.
Our second approach was based on the UCSD p-System [3], only with C as the base and not Pascal. If we could not compile our FORTRAN using existing FORTRAN compilers, and if we could not use the available C compilers directly, then perhaps we could design a C-based pseudo-machine. The FORTRAN codes could be compiled into a pseudo-code—a stack-oriented reverse polish notation. This pseudo- code could then be executed via C. This approach worked. By going to pseudo-code, it was possible to correctly process FORTRAN codes. The system produced the right results, but it was agonizingly slow. And since the entire runtime library had to be linked with the p-machine the executable was very large—this was not the way to go.

The solution came when we discovered the discussion in [2] about the re-creation of source code from reverse polish notation. Using this approach we could compile our codes using the p-machine compiler developed above, and then we could re-create its source code. Only the source code re-created was now C and not FORTRAN.

The Origins of the Approach

The approach taken in the current version of PROMULA.FORTRAN is based on the standard theories of compiler design. Waite and Goos in [1] define a compilation as "a sequence of transformations (SL,L1), (L1,l2),...(Lk,TL) where SL is the source language and TL is the target language." The languages L1 .. Lk are referred to as "intermediate" languages. PROMULA.FORTRAN, then, is a compiler whose source language is FORTRAN, whose intermediate language is a reverse polish pseudo-code, and whose target language is C.

The problem with the standard theories of compiler design is that though there is ample discussion of how to do the conversion from the source language to the intermediate language, there is little discussion of how to move from an intermediate language to a higher level target language such as C. For compilers, the target language is always a low-level machine code. Fortunately, there has been some work done in the area of the re-creation of source code from intermediate code. Many BASIC interpreters need to be able to execute BASIC statements efficiently; while still giving the user access to the source code. This is a problem also faced by contemporary spreadsheet systems. The re-creationists solve this problem by compiling the source statements into a reverse polish notation and then rewriting the source code from that internal notation. An excellent discussion of this work can be found in [2].

The technique of source code recreation from reverse polish notation hinges on the observation that executing such notation via a stack-oriented pseudo-machine is identical to writing and combining the character sequences that perform those operations in some target language via a stack-oriented string manipulation machine! The above can best be understood via a simple example.

Consider the following statement which we wish to both execute via a p-machine and re-create:

```
        A = B * C + D.
```

The intermediate form of this statement might be as follows:

```
        PUSHADR A
        PUSHADR B
        GETVAL
        PUSHADR C
        GETVAL
        MULT
        PUSHADR D
        GETVAL
        ADD
        PUTVAL
        EOS
```

where for the execution pseudo-machine:

```
       Op Code Meaning
       PUSHADR means push the address of the variable onto the stack.
       GETVAL  means pop the address from the stack, obtain the value at that
      address, and push it onto the stack.
       MULT means pop the top two values from the stack, multiply them
      together, and push the result onto the stack.
       ADD  means pop the top two values from the stack, add them together,
      and push the result onto the stack.
       PUTVAL  means pop the value and address from the stack and then store the
      value at the address.
       EOS  means end the current statement.
```

and for the string-manipulation machine:

```
       Op Code Meaning
       PUSHADR means enter the notation for a pointer to the variable onto the
      stack.
       GETVAL  means pop the top string from the stack, convert it to the notation
      for a value at the indicated address, and push that string back
      onto the stack.
```

```
 MULT means pop the top two strings from the  stack, concatenate them
with the symbol '*' in the middle, and push that string back onto
the stack.
 ADD  means pop the top two strings from the stack, concatenate them
with the symbol '+' in the middle, and push that string back onto
the stack.
 PUTVAL  means pop the top two strings from the stack, convert the lower
one to a value string, concatenate the two strings with the symbol
'=' in the middle, and push the resultant string onto the stack.
 EOS  means write the current string.
```

Let us execute these two machines and watch the symmetry between them. Assume that A is stored at address 45, that B is at address 50 and has value 2.0, that C at 55 has value 3.0, and D at 60 has value 4.0. It is both exciting and interesting to note that these address and value assumptions are needed for the execution pseudo-machine only. Alternately, the string manipulation machine needs access to the symbol table, which the execution-machine does not.

```
Instruction  Execution stack  String-manipulation stack
PUSHADR A 45"&A"
PUSHADR B 45, 50  "&A", "&B"
GETVAL 45, 2.0 "&A", "B"
PUSHADR C 45, 2.0, 55"&A", "B", "&C"
GETVAL 45, 2.0, 3.0  "&A", "B", "C"
MULT45, 6.0 "&A", "B*C"
PUSHADR D 45, 6.0, 60"&A", "B*C", "&D"
GETVAL 45, 6.0, 4.0  "&A", "B*C", "D"
ADD 45, 10.0"&A", "B*C+D"
PUTVAL"A=B*C+D"
EOS
```

Though this simple example only shows how to translate "A=B*C+D" back into itself, its extension to approximately 200 pseudo-operations, and its generalization via a "surface-form" notation, allowed us to construct a very fast and sophisticated translator. With a single extension for dealing with operator hierarchies, to be discussed below, the above is all that PROMULA.FORTRAN does. It produces pseudo-code and a symbol table from the source code, simplifies the code by using an execution machine, and then writes the code back out in user-definable C form via a string manipulation machine. A serious problem faced by the re-creationists, is that the production of the output string is completely independent of the original input made by the user. Though the original and the result mean the same thing, they might look very different. Fortunately, in the application to translation this is not a problem. The user wants semantic and not syntactic identity. The greatest weakness of the re-creation approach becomes the greatest strength of PROMULA.FORTRAN.

The Problem with Parentheses

A problem which was ignored in the above example has to do with parentheses or operator precedence. The reason reverse polish notation is typically used as an intermediate language and even in the design of contemporary machine languages, is that it uses no parentheses. Consider the statement: A = B * (C + D) in which the addition operation is to be performed prior to the multiplication. Using the same notation as above the intermediate form of this statement would be as follows:

```
PUSHADRA
PUSHADRB
GETVAL
PUSHADRC
GETVAL
PUSHADRD
GETVAL
ADD
MULT
PUTVAL
EOS
```

Note that the effect of the parentheses in intermediate form was simply to reorder the operations.
Now using the same execution and string-manipulation rules as before the following would result:

```
Instruction  Execution stackString-manipulation stack
PUSHADR A 45 "&A"
PUSHADR B 45, 50"&A", "&B"
GETVAL 45, 2.0  "&A", "B"
PUSHADR C 45, 2.0, 55 "&A", "B", "&C"
GETVAL 45, 2.0, 3.0"&A", "B", "C"
PUSHAD 45, 2.0, 3.0, 60  "&A", "B", "C", "&D"
GETVAL 45, 2.0, 3.0, 4.0 "&A", "B", "C", "D"
ADD 45, 2.0, 7.0"&A", "B", "C+D"
MULT45, 14.0 "&A", "B*C+D"
PUTVAL "A=B*C+D"
EOS
```

Obviously the execution result of 14.0 is correct, but the string-manipulation result of "A=B*C+D" , which is identical to the previous result, is wrong. The parenthesis have been omitted. The re-creationists solve this problem by entering special markers into the intermediate notation to indicate where the parentheses occurred. This technique obviously does not work in the translation application, because the placement of the parentheses in the target language is a function of the operator precedence of the target language and not of the source language.
The solution taken in PROMULA.FORTRAN is quite simple and general. Each rule in the string manipulation machine has associated with it a precedence code. Whenever the output of an operation is written to the string-stack, the precedence code of the rule that produced the output is also written. Whenever an element is combined within a rule, if its precedence code is nonzero and lower than the code of the rule, then that element is enclosed in parentheses as it is concatenated into the new output.

Applying this extension to the example above, the precedence codes for the various operations are as follows:

Operation	Precedence
PUSHADR	0
GETVAL	0
MULT	2
ADD	1
PUTVAL	0

Note that operations not involved in evaluations have a precedence of zero. MULT has a higher precedence then ADD because in C multiplication is higher in precedence than addition. Using the precedence rule from above we can again run the string-manipulation machine on the intermediate form of A=B*(C+D). The rule and stack precedence codes are shown in parentheses.

```
 Instruction String-manipulation stack
 ---------------------------------------
(0) PUSHADRA (0) "&A"
(0) PUSHADRB (0) "&A", (0) "&B"
(0) GETVAL(0) "&A", (0) "B"
(0) PUSHADRC (0) "&A", (0) "B", (0) "&C"
(0) GETVAL(0) "&A", (0) "B", (0) "C"
(0) PUSHADRD (0) "&A", (0) "B", (0) "C", (0) "&D"
(0) GETVAL(0) "&A", (0) "B", (0) "C", (0) "D"
(1) ADD(0) "&A", (0) "B", (1) "C+D"
(2) MULT  (0) "&A", (2) "B*(C+D)"
(0) PUTVAL(0) "A=B*(C+D)"
 EOS
```

The critical point in the above occurs when the output of the ADD operation, which has a precedence code of 1, is combined via the MULT operation which has a precedence of 2. Since 1 is nonzero and less than 2, the string "C+D" is enclosed in parentheses as it is entered into the output string of the MULT instruction.

The Translation Algorithm

To understand the "surface-form" description language, an overview of the entire translation algorithm is first needed. This algorithm has three requirements:

1. The source language must be translatable into an intermediate language all of whose operators are expressed in reverse polish notation and be they unary, binary, ternary, etc.—produce either a single result or no result.
2. In the target language every n-ary operator can be expressed as an arbitrary but fixed concatenation of its operands and of other fixed character sequences.
3. It must be possible to gain access to all operations performed in the source language via the target language.

The first two requirements are easy to achieve via almost any contemporary language. The third is the killer. We were asked at one time, for example, if we could translate C into FORTRAN. The answer was "no" because C pointer operations cannot be expressed in FORTRAN.
The actual application of the requirements will become clearer in the following, though emphasis will be placed on target language production. Remember also that the world is not perfect. The algorithm works, but there are places where special processing is needed, especially in the area of the data definition component. Promula. FORTRAN is not a universal translator -it is a translator from FORTRAN to C; however, it is sufficiently general to give the wide range needed to deal with the many FORTRAN dialects and the many user output biases.

The Components of a Translation Definition

A complete translation definition contains seven major components of which the first five are general and accessible to the end-user and two are hardwired in the translator. They are as follows:

1. A definition of the basic operation codes which make up the pseudo- machine. These definitions form the glue which ties the other components together.
2. A definition of how the expressions of the language are broken down into operation codes. This component contains five subcomponents:
    1. The operations needed to perform type conversions ·
    2. The promotion hierarchies used when deciding what type conversions to perform ·
    3. The unary operators ·
    4. The binary operators ·
    5. The functions available
3. A definition of the actual statements in the source dialect. Within this component statements are classified by type. Each type contains its own additional information requirements. Both data definition and executable statements are included in this component.
4. A description of the C surface-form to be taken by each operation. These surface-form descriptions look much like the control strings used by the C printf function, except that the conversion symbols refer to the elements on the translation stack.
5. A runtime library which performs those operations referenced in the target language but not implemented directly.
6. A set of hard-wired functions built into the translator which process the basic input structure of the source language and which build the various symbol tables.
7. A set of hard-wired functions built into the translator which process the basic output structure of the target language and which produce the needed data definition statements from the symbol tables.

As was said above the basic flavors of FORTRAN and C are built into the translator via the last two components. These cannot be altered by the end-user. It is this hard-wiring that makes the translator fast and compact.
When new users first look at the PROMULA.FORTRAN translation definition file, they are usually suprised to see what appears to be just a long list of operation code identifiers. It is, however, in terms of these identifiers that the entire translation process is tied together. All executable FORTRAN statements are translated into these operation codes, and all executable C output statements are described in terms of these codes. PROMULA.FORTRAN is ultimately a pseudo-code compiler.

Via the second and third components, the user has considerable flexibility in defining his FORTRAN source dialect. Not much will be said about the notation used for describing the source dialect. It is interesting to note, however, that traditional language descriptions lump expression description in with statement descriptions. We have found that processing is much more efficient if the description of and, therefore processing of, expressions is separated from statement description and processing. Thus, component (2) describes expressions, while component (3) describes how those expressions are combined to form statements.

Components (4) and (5) will be discussed throughout the remainder of this paper. The actual surface-form description strings are contained in (4) and any functions referenced via those surface form description strings are contained in (5). Potential users of PROMULA.FORTRAN occasionally express surprise at the runtime library. Why, if we are translating into C, do we need a runtime library? To them it seems like cheating. Only C functions should be used if a true translation is achieved. Clearly, there are operations performed in FORTRAN which are not directly accessible in C -complex arithmetic, FORTRAN style formatted I/O, etc. These operations must be performed via runtime functions, but the structure of these functions is largely determined by how the translation is performed. Thus, components (4) and (5) are closely tied.

The Surface-Form Description Language

For each operation code in the intermediate language, there is an entry in the surface-form description language. Each operation description has three components which may vary by the user output bias type:

1. A specification of the number of operands associated with the operation -- i.e., whether the operation is null, or unary, or binary, etc. -- for the particular output bias. Remember that all operations are reverse polish; therefore, when a given operation is encountered, its operand strings have already been placed on the string-machine stack. The operands are numbered starting with the oldest first. In other words, the operand deepest on the stack is argument 1 and the operand at the top of the stack is argument n, for an n-ary operator.

2. A specification of the precedence of the operator relative to others. As the output production proceeds, it is necessary to enclose certain operations in parentheses to achieve the proper order of evaluation. The current precedence of each operand is maintained. When two operators of lower precedence are combined via an operator of higher precedence, then they are enclosed in parentheses.

3. A pattern string which specifies an arbitrary but fixed con- catenation of the operands and of other character sequences which can be described via a linear pattern string. This pattern string is, of course, the actual implementation of our re-creation algorithm requirement number (2) as presented earlier. The pattern string describes not only how the operands are combined but also how the various constants, symbol table entries, and miscellaneous special-purpose conversion routines combine to form the final output.

The bulk of the discussion below deals with pattern strings. They are deceptively simple.

The Example Revisited

Earlier a set of six operation codes was presented along with a verbal description of what each did when executed via the string-machine. This subsection presents the identical information using the surface-form notation in order to introduce the concept. The actual specification is as follows:

```
SURFACE-FORMS
PUSHADRPATTERN 0, 0, "\v"
GETVAL PATTERN 1, 0, "%1i"
MULTPATTERN 2, 2, "%1d * %2d"
ADD PATTERN 2, 1, "%1d + %2d"
PUTVAL PATTERN 2, 1, "%1i = %2d"
EOS PATTERN 1, 0, "%1d\c"
END
```

As specified above, the first number parameter specifies the number of arguments and the second the precedence code. The string is the output pattern string. The basic operation of the pattern strings is straightforward. The output processor moves to the next pseudo-code in the intermediate language. It then looks up the pattern information for that code. It removes as many operand strings from the string stack as are specified in the first parameter of the specification. It saves pointers to these operand strings and the current precedence associated with each in a temporary buffer. Next, a new string is formed using the actual pattern string as a guide. Finally, the result string is pushed onto the stack and assigned the hierarchy specified in the second parameter.

Examining the pattern strings themselves, notice first that they resemble those used by C to specify output conversions. This similarity is deliberate, since the purpose of these strings is in essence the same. Within the pattern strings there are three types of specifications:

1. Special operation parameters which consist of a backslash followed by a letter. These parameters trigger special conversions. Thus, in the above rule for PUSHADR the \v specifies that a pointer to a variable whose symbol number is specified following the opcode is to be entered into the output string at the indicated location. The \c notation in the EOS string specifies that the current string stack is to be cleared and thus written to the output file.

2. Operand conversion parameters which consist of a percent sign, followed by a numeric digit, followed by a conversion code. The numeric digit specifies which operand is to be entered at this point in the string, and the conversion code specifies any special operation to be performed. In the above, the %1d which occurs in the MULT, ADD, and EOS strings says "Enter the first operand without any additional editing (other that any precedence editing which may be needed) into the output string." The %2d says the same thing for operand 2. The %1i specification in GETVAL and PUTVAL says "Enter the string in instantiated form -- i.e., change it from a pointer representation to the representation of the value pointed to."

3. Simple character specifications are any characters not forming one of the two specifications above. Simple characters are entered into result strings exactly as entered.

This is basically all there is to know about pattern strings. There are obviously more specification codes; however, their description here is not important. As can be appreciated, the above notation is extremely powerful; especially when combined with the notion of user output bias.

Accounting for User Bias

As was discussed in the introduction different users have different uses for the output of the translator. In particular, some want optimized code, some want C-like maintainable code, and some want FORTRAN-like code. This general issue is referred to as the user bias. To deal with this problem, the surface-form description language allows separate entries for each user bias. The next section will deal with some input-output statements in great detail. To introduce the topic let us look once more at the simple statement: A = B * (C + D). Let us now pretend that under certain circumstances the variables A, B, C, and D are complex. In this case, the complex functions cadd and cmul perform the arithmetic operations. We will assign this "complex" condition a bias code of C. Now, the surface-form description is as follows:

```
SURFACE-FORMS
PUSHADRPATTERN 0, 0, "\v"
GETVAL PATTERN 1, 0, "%1i"
MULTPATTERN
 C  2, 0, "cmul(%1d, %2d)"
 *  2, 0, "%1d * %2d"
ADD PATTERN
 C  2, 0, "cadd(%1d, %2d)"
 *  2, 1, "%1d + %2d"
PUTVAL PATTERN 2, 1, "%1i = %2d"
EOS PATTERN 1, 0, "%1d\c"
END
```

Whenever a given operation has an alternate form for one or more bias codes, the separate entries are entered on separate lines, preceded by the bias code. The '*' code is always last and specifies the default pattern. Note that the precedence codes are different between the two output biases. In a later example, even the number of parameters will differ.

Executing the string-machine using these two alternate biases looks as follows:

```
Instruction  Complex biasDefault bias
------------------  -------------------
PUSHADRA  "&A"  "&A"
PUSHADRB  "&A", "&B"  "&A", "&B"
GETVAL "&A", "B""&A", "B"
PUSHADRC  "&A", "B", "&C""&A", "B", "&C"
GETVAL "&A", "B", "C" "&A", "B", "C"
PUSHADRD  "&A", "B", "C", "&D" "&A", "B", "C", "&D"
GETVAL "&A", "B", "C", "&D" "&A", "B", "C", "D"
ADD "&A", "B", "cadd(C,D)"  "&A", "B", "C+D"
```

```
MULT"&A", "cmul(B,cadd(C,D))"  "&A", "B*(C+D)"
PUTVAL "A=cmul(B,cadd(C,D))""A=B*(C+D)"
EOS
```

Note that though the translations look quite different, the code generated for the two biases is identical. The C output is controlled entirely via the the surface-form description language.

A Sample Translation

This section gives a complete example, using the FORTRAN OPEN statement, of a translation and of the considerations needed to produce that translation for the various user output biases. The material presented shows the actual specifications and output results as produced by the current version (1.22) of PROMULA.FORTRAN. Before beginning this discussion it must be emphasized that though any user of PROMULA.FORTRAN has full access and control over the following specifications, the typical user accesses them in compiled form only and needs only specify his overall output bias.

The OPEN statement is selected because its operation should be clear to those not familiar with FORTRAN, while its syntax and implementation are sufficiently complex to make the points needed.

The FORTRAN OPEN Syntax

In this presentation a slightly simplified version of the OPEN statement is presented. The purpose of the OPEN is similar to the C "fopen" function though it cannot be directly translated into "fopen". The primary problem is that in FORTRAN the user assigns arbitrary integer handles -- referred to as "logical unit numbers" -- which he then uses to reference the file. The handle is not assigned by the runtime system.
The syntax of the OPEN is as follows:

```
OPEN ( [UNIT=] u [,IOSTAT= ios] [,ERR= sl], [,FILE= fin]
 [,STATUS= sta] [,ACCESS= acc] [,FORM= fm]
 [,RECL= rl] [,BLANK= blnk] )
Where:

      u  is an integer specifying the unit number
      iosis an integer receiving any error codes that occur
      sl is the label to branch to on an error
      stais a string specifying the file status
      accis a string specifying the access type
      fm is a string specifying the record format
      rl is an integer specifying the record length
```

The Runtime Library Structure

Suffice to say that the FORTRAN file system is clearly different than the standard C file system; therefore, a basic set of utilities is needed in the runtime library to deal with the OPEN statement. These are as follows:

```
void fiostatus(iostat,error)
long* iostat;Address of error status variable
int error;Error testing switch
Description:  If the FORTRAN I/O runtime system encounters an error, it
sets an error code and calls function "fioerror". The behavior of that
function depends upon how the code using the I/O system is doing error
processing.  This function establishes the error code return variable and
the error testing switch.  If that switch is zero, then an error on a
successive I/O operation causes an abnormal termination.  If the switch
is nonzero, then an error condition is set and a normal return is executed.
int fiolun(lun,action)
int lun; Logical unit number of file
int action; Action code for subsequent use
Description:  Before any action can be performed on a FORTRAN file, the
logical unit number must be associated with an existing FORTRAN file
structure.  If there is no already existing structure for the unit number,
then this function will attempt to create one.  The form of this creation
depends upon the type of the action to be performed.  These will not be
detailed here.  For OPEN, the access code is zero.
```

```
void fiofdata(option,string,ns)
int option; Specifies which data is being specified
char* string;  String information
int ns;  String length or integer information
Description:  This function is used to specify the various file data
options associated with the current FORTRAN file structure.  The particular
data being specified is defined by the "option" parameter.  These will not
be detailed here.
int fioopen()
Description:  This function opens the file associated the "current" FORTRAN
file using the current specifications as established via previous calls to
function "fiofdata".
int fioerror(clear)
int clear;Should error control be cleared?
Description:  If the FORTRAN I/O runtime system encounters an error, it
sets an error code and calls this function. This function either sets an
error return value or exits to the operating system with an error message.
In the case where an error code is returned to the calling function, the
parameter "clear" specifies whether or not the error processing control
variables should be cleared prior to the return.
```

The Optimized Translation

Given the above syntax and the above runtime library structure the optimal translation of the following FORTRAN fragment

```
SUBROUTINE DEMO
INTEGER ERRCOD
OPEN(1,STATUS='NEW',FILE='demo.out',IOSTAT=ERRCOD,ERR=900)
OPEN(2,STATUS='OLD',FILE='demo.inp',ACCESS='DIRECT',RECL=100)
  900 RETURN
END
```

is as follows:

```
#include "fortran.h"
void demo()
{
static long errcod;
 fiostatus(&errcod,1);
 fiolun(1,0);
 fiofdata(1,"demo.out",8);
 fiofdata(2,"NEW",3);
 fioopen();
 if(fioerror(1)) goto S900;
 fiolun(2,0);
 fiofdata(1,"demo.inp",8);
 fiofdata(2,"OLD",3);
 fiofdata(3,"DIRECT",6);
 fiofdata(5,NULL,100);
 fioopen();
S900:

 return;

}
```

First the fiostatus function is called if the user is performing his own error processing. Next, the fiolun function is called to establish the logical unit number of the file, and the fiofdata functions are called to establish the various parameter values. Finally, the actual open operation is performed and if so requested a possible branch is taken if an error occurred. The above translation is optimal in that it gets the job done in the most efficient and straightforward way. It is the translation for those users who have no direct interest in the intermediate C output.

The remainder of this subsection shows how the above translation is achieved and alternate translations can be produced which are less optimal but which are more readable and more maintainable.

The OPCODES Component

The first step in any translation specification is the establishment of the intermediate language and of its operation codes. These codes tie the translation specifications together. The basic operations involved in the OPEN are as follows:

Opcode	Description
LDA	Load the address of a variable onto the stack
LIC	Push an integer constant onto the stack
STA	Establish the current eror processing status
LUN	Specify the logical unit number
LSC	Load a string constant onto the stack
DFA	Define file access type
DFN	Define current file name
DFR	Define file record length
DFS	Define current file status
DSP	Push a dummy string parameter onto the stack
DIP	Push a dummy integer parameter onto the stack
OPN	Open the current file
ERR	Perform error testing
NER	End without error processing
JMC	Jump on condition
GTL	Go to line address

The LDA, LIC, LSC, DIP, and DSP operation codes are generic operations that place values on the stack. The ERR, NER, JMC, and GTL operations are the branching control operations needed to define the error branching. The STA, LUN, DFA, DFN, DFR, DFS, and OPN operations are particular to the operations of FORTRAN I/O processing.

It must be emphasized again that PROMULA.FORTRAN is a p-code compiler. These operation codes were designed to make it possible to execute FORTRAN programs via a pseudo-machine. The translation to C works completely independently of the compilation to the pseudo-code.

The OPEN Statement Description

To illustrate the point that the transformation to the intermediate opcodes as listed above is independent of the output production, below is the actual statement description for the OPEN statement as entered in the PROMULA.FORTRAN translation description file.

```
 "OPEN"  "(" ["UNIT="] u [ ( "IOSTAT=" ios) | ( "ERR=" sl ) |
  ( "FILE=" fin ) | ( "STATUS=" io_sta ) | ( "ACCESS=" acc ) |
  ( "FORM=" fm ) | ( "RECL=" rl ) |
  ( "BLANK=" blnk ) ]... ")" .
 Where:

      u = integer_parameter
      ios = long*
      sl = statementlabel
      fin = string
      sta = string
      acc = string
      fm = string
      rl = integer_parameter

 Emissions:

      IF(ios & sl) EMIT(ios LIC 1 STA)
      ELSEIF(ios) EMIT(ios LIC 0 STA)
      ELSEIF(sl) EMIT(LDA 0 LIC 0 STA)
      EMIT(u LIC 0 LUN)
      IF(fin) EMIT(fin DFN)
      ELSE EMIT(DSP)
      IF(ios) EMIT(ios DFS)
      ELSE EMIT(DSP)
      IF(acc) EMIT(acc DFA)
      ELSE EMIT(DSP)
      IF(fm) EMIT(fm DFF)
      ELSE EMIT(DSP)
      IF(rl) EMIT(rl DFR)
      ELSE EMIT(DIP)
      IF(blnk) EMIT(blnk DFB)
      ELSE EMIT(DSP)
      EMIT(OPN)
      IF(sl) EMIT(ERR JMC 5 GTL sl)

 End
```

Very briefly, statement descriptions consist of four parts:

1. The left-hand-side recognition symbol. In this case the OPEN statement is recognized when the token "OPEN" is found at the beginning of a statement.

2. The actual syntax description, which consists of formal symbols, such as ( ) [ ] { } . |, etc., terminal symbols enclosed in quotes, and nonterminal symbols. The syntax is designed to look as much as possible like the syntax usually used in FORTRAN reference manuals, but formalized to make it equivalent to contemporary Backus-Naur notation.

3. The "where" section which specifies what each nonterminal symbol is. On the right-side of these specifications is either a type specifier -- meaning an expression of the indicated type -- or the identifier of another statement component description defined like the above.

4. The "emission" specification which specifies how the actual intermediate code is to be structured.

We are omitting from this discussion the description of the basic expressions, which uses an entirely different notation. The linkage between expressions and statements is made in the "where" section above.

The bulk of the above notation should be at least readable by those who are used to formal language descriptions; however, the emission section is unique to PROMULA.FORTRAN. Its purpose is to specify how the actual code sequences associated with the nonterminal symbols are to be combined in the context of this statement. In the conditional portions of the notation, a nonterminal symbol is true if it has any code associated with it, and false if not. Testing a conditional does not effect its code. In the EMIT clauses constants and operation codes are emitted exactly as entered. For nonterminal symbols, the code associated with that symbol is emitted. This notation is very powerful and is able to deal with must of the FORTRAN syntax -- in particular the I/O statements.

The Biased Surface-Form Descriptions

Once statements have been processed into the intermediate language as defined by the opcode list using specifications of the statement syntax and emissions, the intermediate language can be output as C using the biased surface-form descriptions. As has been discussed before, there are three distinct views of how the C output should look.

The first bias, wants an optimal C which is quick to compile and which minimizes additional runtime code. The translation for this bias was presented earlier in order to introduce the runtime library. The translation gives an optimal use of this library with no additional overhead.

The second bias, wants a C code that can be easily read and maintained, but which follows C conventions as closely as possible. Our default translation under this bias is shown below.

```
 #include "fortran.h"
 void demo()
 {
 static long errcod;
  fiostatus(&errcod,1);
  ftnopen(1,"demo.out",8,"NEW",NULL,NULL,0L,NULL,0L);
  if(fioerror(1)) goto S900;
  ftnopen(2,"demo.inp",8,"OLD","DIRECT",NULL,100,NULL,0L);
 S900:

return;

 }
```

The error processing logic is the same as before, but an additional interface routine "ftnopen" is introduced which combines all of the possible parameters to the OPEN into a single function call with a fixed parameter list. This function itself then calls the fioopen and fiodata functions based on the parameter list. This implementation is clearly easier to read, but it requires slightly more runtime code.

The third type of user wants to retain as much of the original FORTRAN flavor as possible. He is used to maintaining the FORTRAN code and he wants to make the transition to C as painless and error-free as possible. For this user, the C output is as follows:

```
#include "fortran.h"
#include "ftnsymb.h"
void demo()
{
static long errcod;
 if(IO_ERR(OPEN(1,O_IOSTAT,&errcod,O_FILE,"demo.out",8,
  O_STATUS,"NEW",0))) goto S900;
 OPEN(2,O_FILE,"demo.inp",8,O_STATUS,"OLD",O_ACCESS,"DIRECT",
  O_RECL,100,0);
S900:

 return;

}
```

Note first that an additional file "ftnsymb.h" is included. This file contains definitions of various constants intended to remind the user of the original FORTRAN. Also a new function OPEN is introduced, which is defined as "fiofopn" in the include file. This function takes a variable number of arguments, each preceded by an integer code with the last argument being zero. The integer codes themselves are represented by the symbols O_IOSTAT, O_FILE, etc., in the include file. In addition, the OPEN function itself returns the error exit condition so that it can be nested in the IO_ERR function which is simply defined as "fioerror" in the include file. The advantage of this translation is that there is a 1-to-1 correspondence between input statements and output statements and that the form of the C is at least reminiscent of the original. The disadvantages are that an additional runtime function is needed and extra time is needed to compile the additional symbols.

Having presented the three biased outputs, all that remains are the surface-form descriptions themselves.

```
 DFS PATTERN
  F  3, 0, "%1d,O_STATUS,%2d"
  C  2, 0, "%1d"
  *  2, 0, "fiofdata(2,%1d,%2d);\c"
 LUN PATTERN
  F  3, 0, "%2d%1d"
  C  2, 0, "%1d"
  *  2, 0, "fiolun(%1d,%2d);\c"
 LSC PATTERN 0, 0, "\ks"
 DFA PATTERN
  F  3, 0, "%1d,O_ACCESS,%2d,%3d"
  C  2, 0, "%1d"
  *  2, 0, "fiofdata(3,%1d,%2d);\c"
 DFN PATTERN
  F  3, 0, "%1d,O_FILE,%2d,%3d"
  C  2, 0, "%1d,%2d"
  *  2, 0, "fiofdata(1,%1d,%2d);\c"
 DFR PATTERN
  F  2, 0, "%1d,O_RECL,%2d"
  C
  *  1, 0, "fiofdata(5,NULL,%1d);\c"
 DSP PATTERN
  C  0, 0, "NULL"
  *
 DIP PATTERN
  C  0, 0, "0L"
  *
 OPN PATTERN
F  1, 0, "OPEN(%1d,0)"
C  8, 0, "ftnopen(%1d,%2d,%3d,%4d,%5d,%6d,%7d,%8d);\c"
*  0, 0, "fioopen();\c"
 DFF PATTERN
  F  3, 0, "%1d,O_FORM,%2d,%3d"
  C  2, 0, "%1d"
  *  2, 0, "fiofdata(4,%1d,%2d);\c"
 ERR PATTERN
  F  1, 0, "O_ERR(%1d)"
  *  0, 0, "fioerror(1)"
 NER PATTERN
  F  1, 0, "%1d;\c"
  C  1, 0, "\c"
  *  0, 0, "\c"
```

As described before, in the above descriptions those operations that have a single set of pattern specifications use the same pattern for all output biases. Those that have multiple patterns use the one specified for a given bias type if present, else they use the last pattern introduced by the '*'. Consider the OPN operation itself.
For the FORTRAN bias the following pattern is used:

```
      1, 0, "OPEN(%1d,0)"
```

In this form, by the time OPN is executed by the string-machine, there is a single pattern string on the stack, which contains the concatenation of the other parameters as they occured. This string is entered into the expression followed by a zero, which terminates the parameter string. The string thus formed is not yet cleared to the output file since it may yet be concatenated into the error function call.

For the C bias the following pattern is used:

```
       8, 0, "ftnopen(%1d,%2d,%3d,%4d,%5d,%6d,%7d,%8d);\c"
```

In this form, by the time the OPN is executed by the string-machine, there are eight independent parameters on the stack. It is interesting, but not really important, that the order of the operators in the string-machine varies wildly depending upon the output bias. The pattern string concatenates the eight parameters into a single function call. The stack is then cleared, since the error processing is performed via a separate statement.
For the optimized bias, the following pattern is used:

```
       0, 0, "fioopen();\c"
```

In this form, the stack has already been cleared prior to the execution of the OPN operation by the string-machine. There are no parameters. The operation itself merely writes the appropriate function call and clears the string stack. The additional pattern strings may be analyzed in the same manner as above.

Other Bias Switches

In addition to the actual statement translations, which are controlled via the surface-form description language, there are several other translation features and optimizations which can be individually controlled by the user. Each has a default setting for the three output biases, though these settings can always be overriden by the user. These are best described via an example. Consider the following FORTRAN subroutine which computes the mean and variance of a set of values along with its optimized bias C translation. All flags are on for the optimized bias.

INPUT	OUTPUT	NOTES
SUBROUTINE EX001(VAL,N,XBAR,VAR)	void ex001(val,n,xbar,var)	
DIMENSION VAL(N)	int n;	(1)
XBAR=0.0	float *val,*xbar,*var;	
VAR=0.0	{	
DO 10 J = 1,N	auto int j;	
XBAR = XBAR + VAL(J)	auto float s;	
10 CONTINUE	*xbar = *var = 0.0;	(2)
XBAR = XBAR/N	for(j=0; j<n; j++) *xbar +=	(3)-
DO 15 J = 1,N	*(val+j);	(6)
S = VAL(J) XBAR	*xbar /= n;	(5)
VAR = VAR + S*S	for(j=0; j<n; j++) {	(3) (4)
15 CONTINUE	s = *(val+j)-*xbar;	(5)
VAR = VAR/(N-1)	*var += (s*s);	(7)
RETURN	}	(5)
END	*var /= (n-1);	
	}	

Note (1) that the parameter "n" is not declared as a pointer, since it is not changed within the routine. promulaFortran uses what are called "prototypes" of subprogram arguments so that it can generate optimal calling sequences. These "prototypes" may be specified by the user or may be determined internally by the translator. The above was internally determined by the translator.

Note (2) that C allows multiple assignments to the same value to be written together. The translator looks for such assignments and combines them whenever possible.

Note (3) that in FORTRAN the default base for a subscript is 1. Thus, all do loops which generate subscripts tend to start at 1. In C, however, subscripts start at zero. This fact makes for much more efficient code. The translator looks for do loops whose only purpose is to move through array subscripts and reduces their range to start at zero, thus producing a very natural looking for-statement and optimizing subscript expressions.

Note (4) that C has "++" and "--" operators which take advantage of the fact that most computers have increment and decrement operators. The translator uses these operators whenever possible.

Note (5) that C has operators like "+=", "-=", "*=", "/=", etc. The use of these operators ensures that the address of the left-hand-side of the assignment will only be computed as often as necessary. promulaFortran uses these operators.

Note (6) that the do loop running to statement 10 in the FORTRAN code is collapsed into a single compound statement, and that the now unneeded statement label is removed.

Note (7) that though the do loop statements in loop 15 cannot be reduced to a single statement, the statement label can still be removed.

Conclusion

The intent of this paper was to give a clear view of a problem faced by translation software which is not faced by other compilers. When the target language is human-readable, then humans want some say in its form. Machines do not care about the syntax of machine- readable target languages. This fact greatly complicates the design of the software. But using the techniques described above, PROMULA.FORTRAN is not only one of the best FORTRAN to C translators available, it is also an excellent FORTRAN compiler.

References

[1] Waite, W. M. and G. Goos *Compiler Construction*. Springer-Verlag, New York, 1984, p. 4.

[2] Brown, P.J. "More on the Recreation of Source Code from Reverse Polish." *Software Practice and Experience*, Vol. 7, p545-551 (1977)

[3] *Internal Architecture Guide for the UCSD p-System (TM), Version IV.0*, IBM Personal Computer Language Series, January 1982.

Fred Goodman is a mathematician/linguist and the author of PROMULA.FORTRAN and Promula, an applications development tool for modeling applications using databases. Fred is also the Vice President and Director of Technical Development at Promula Development Corporation. He is currently applying the translation methodology discussed here to BASIC, PASCAL, and COBOL as well as other special-purpose languages on multiple platforms.