A FORTRAN PREPROCESSOR FOR THE LARGE PROGRAM ENVIRONMENT

Neal R. Wagner
Computer Science Department
University of Houston
Houston, Texas 77004

Abstract. The use of a preprocessor to aid structured programming in Fortran has been widely discussed. This article considers a design philosophy which is especially oriented toward large program development and maintenance. The design is distinguished by the retention of the form of the original source program in the standard Fortran output by the preprocessor. A specific implementation is described.

I. INTRODUCTION

Various software tools are often nearly essential for large program development, and one such useful tool is a preprocessor for the language used. A preprocessor is especially helpful for any development involving the Fortran language as supplied by many vendors. Even after Fortran 77 is widely implemented, eliminating some of the need for a preprocessor, requirements on a large programming project will often be so specialized that they will best be met by a preprocessor written specifically for that one application. (Usually the applications programmers themselves should design and implement such a preprocessor.) One can achieve the effect of a change in the compiler by a change in the preprocessor.

This article focuses on a specific experimental Fortran preprocessor. However the article also presents a much more broadly applicable design methodology, namely, a preprocessor which incorporates the original source text into its output.

II. DESIGN GOALS AND MOTIVATION

There now exist a surprising number of special Fortran preprocessors. They serve to extend the Fortran language with new control structures or new data types or other features. (See Reifer [12] for a list of 55 such preprocessors.) These preprocessors often suffer from many of the following disadvantages:

Fortran Preprocessor

1. Compile-time and execution errors are not described in source language terms.

2. The preprocessed form of the program is hard to read and understand, and hard to correct or modify.

3. There is a need to maintain two separate listings because of items 1 and 2.

4. In some systems the extended features being processed are not checked for errors.

5. Each preprocessor implements different types and forms of language extensions.

6. The preprocessor uses up extra execution time.

7. The preprocessed text suffers in comparison to hand-coded text designed to do the same task.

For example the RATFOR preprocessor developed by Kernigham [8], though very successful, suffers from all the disadvantages except item 4.

In the large program environment, items 1, 2 and 3 form the heart of the problem. (In an environment consisting mostly of smaller programs item 6, the extra execution time, may be the most important, as is discussed in the context of WATFIV-S by Dirkson and McPhee [4].) When the author proposed to several colleagues in an industry environment that a preprocessor be used for developing a large collection of Fortran software, their main reason for rejecting the idea was the need for two separate listings, since this would greatly increase the amount of work required during debugging and maintenance.

With this criticism in mind, the author began the design of a preprocessor which would include the original source language as comments in the proprocessed text. If this were done in such a way that the original source was easily readable then only a compiler listing of the preprocessed text would be needed.

A number of authors have suggested similar systems, the closest being that of Boddy [1]. Other interesting proposals have been given by Bond [2], Charmonman and Wagener [3], Gales [5], Higgins [6], Horowitz [7] and Myers [10]. However, none of these authors mention a key feature of our system: the ability to recover the original source code from the preprocessed text. This feature will be discussed more thoroughly later.

Of course the idea of including the original source is not new. Assembler listing produced by certain compilers give the compiled high-level statements interspersed in the assembler code. Another example is the inclusion of macro expansions in assembler code.

Fortran Preprocessor

Notice that there are other posssible ways to try to eliminate the need for double listings. For example, one could in a variety of ways retrieve the line numbers from the compiler listing and match them up with the appropriate statements in the original source listing. In fact, our system also does this, as will be described in the next section.

Our original design is illustrated in Figure 1, which gives the preprocessed text. There is no need to give the source text, since the source is what appears between the two dashed vertical lines. (Actually, we have taken certain liberties here, since Figure 1 is part of a listing produced by an advanced version of the preprocessor. The original design differed in a few minor ways.) The basic design strategy, as should be evident from Figure 1, was to shield the extended statements as comments and to move the generated standard Fortran to the far right 15 columns. Vertical lines are inserted where possible in columns 6 and 57 to separate out the original source. In addition to the features illustrated, the original design had an "INCLUDE" statement for incorporating source text files.

Simplicity was also an important part of the original design. For example, such "syntactic sugar" as the use of "!=" as an alternative to ".NE." in the RATFOR system [8] was resisted, since too many such special features can make it harder to train newcomers to the system.

Finally, the design aimed to be as flexible as possible so that many different combinations of features could be tried out. This preprocessor was always visualized as an experimental tool, and during its development many changes were tried out in the syntax of the extended statements, in the way they were shielded as comments, in the form of the generated standard Fortran, and so forth. In this sense Figure 1 represents a fairly late stage in the development.

```
C#   !FOR S FROM T TO 1 BY -1 DO                         !
                                                             I90007=(1-T
     !                                                       +(-1))/(-1)
                                                                S=T-(-1)
90007                                                           S=S+(-1)
                                                            IF(I90007.LE
     !                                                      .0)GOTO90009
        H = HT(S)
C#   !  FOR J FROM H+1 TO N DO                           !
                                                                I90010=N
     !                                                          -(H+1)+1
                                                               J=(H+1)-1
90010                                                              J=J+1
                                                            IF(I90010.LE
     !                                                      .0)GOTO90012
           I = J - H
           RR = R(J)
C#   !     WHILE RR .LT. R(I) DO                         !
90013                                                    IF(.NOT.(RR.LT.
     !                                                   R(I)))GOTO90015
              R(I + H) = R(I)
              I = I - H
              IF (I .LE. 0)                                   GOTO 90016
C#   !        THEN                                       !
                                                              GOTO 90017
90016                                                           CONTINUE
C#   !           BREAK                                   !
                                                              GOTO 90015

C#   !        FI                                         !
90017                                                           CONTINUE
C#   !     OD                                            !
90014                                                         GOTO 90013
90015                                                           CONTINUE
           R(I+H) = RR
C#   !  OD                                               !
90011                                                    I90010=190010-1
                                                              GOTO 90010
90012                                                           CONTINUE
C#   !OD                                                 !
90008                                                    I90007=I90007-1
                                                              GOTO 90007
90009                                                           CONTINUE
      RETURN
      END
```

FIGURE 1. Preprocessed text.

III. FINAL DESIGN AND IMPLEMENTATION

Rather than dwell on this preprocessor's evolution, we will discuss its final form in this section. The next section will discuss the development and use of the preprocessor.

The final version of the preprocessor consists of two PL/I programs: the preprocessor itself (over 1000 card images long) and a separate postprocessor for compiler listings (about 150 card images long). There are also four catalogued procedures (collections of control cards) for using these programs.

One of the most interesting features of this preprocessor is that it will run in two modes: "normal" and "recover". In "normal" mode, comments and standard Fortran are ignored, and the language extensions are translated as one would expect. If part of the source text contained text that had already been run through the preprocessor, this source text would look like comments and standard Fortran and so would be ignored.

In "recover" mode, first the original source is reconstructed and then the preprocessor proceeds as in "normal" mode. If no part of the source text has already been preprocessed, this mode is equivalent to "normal" mode. If part (or all) of the source text had been preprocessed, the effect of the preprocessing on that part would be undone, so that preprocessing would start from scratch on all parts of the source text.

A "recover" mode implies an algorithm for reconstructing the source text from the preprocessed text. The algorithm is clear from Figure 1 and from the following restrictions on the form of standard Fortran that a user of the preprocessor must employ.

1. A user is not allowed to put a "#" character in column 2 of a comment. Thus the preprocessor can assume that any comment starting with "C#" is generated by itself and can easily reconstruct the original source that the comment shields.

2. A user is not allowed to code a standard (non-comment) Fortran statement with blanks in columns 7-57. Thus the preprocessor knows that any standard Fortran statements which are not comments and which use only columns 1-6 and 58-72 are statements which it generated. In "recover" mode such statements can be recognized and deleted. (In practice, few people would ever space over to column 58 to begin a Fortran statement.)

All communication to and from the preprocessor is via Fortran comments. Any options to the preprocessor are specified on a special "C OPTIONS" card at the beginning of the program. The "C OPTIONS" card is itself preprocessed (or generated if not present) to indicate also the default options in effect. The preprocessor identifies itself by two comments and produces "error message" comments for any errors it detects. Finally it produces an error count for each subprogram and a grand total of the number of errors. (See Figure 3 for examples.)

Let us list and discuss some of the preprocessor options as given on the "C OPTIONS" card at the head of Figure 2. The effects of several of these options are shown in Figure 2.

1. As discussed above, the preprocessor will run in either "normal" or "recover" mode, where "R=1" means "recover" mode.

2. The starting value for the 5-digit integers used for generated statement labels can be changed from the default value given by "N=90001". (This is essential to avoid duplicate labels if we add extended statements to preprocessed text and run the preprocessor a second time in "normal" mode.)

3. The preprocessor will produce a source listing ("S=1"), a listing of the preprocessor output ("S=2") or a "recovered" source listing ("S=3"). (Usually one would want none of these, but a "recovered" compiler listing as shown in Figure 2 and discussed in the next section.)

4. The number of columns (right-justified to column 72) for the generated standard Fortran can be changed from the default value given by "L=15".

5. The character used to mark boundaries can be changed from the default given by "B='¦'". (If B is set equal to a blank, then the boundaries are not marked.)

6. If requested, the preprocessor will automatically reformat source text to provide uniform amounts of indentation for blocks and loops. This is specified by setting "F" equal to some positive integer which will be the number of columns used for each level of nesting. Comments are only reformatted if the first non-blank character after column 6 is a "*".

7. The preprocessor will optionally put grid marks of "." into the reformatted comments to indicate the levels of nesting.

8. The preprocessor will produce update numbers for columns 73-80 of generated statements. (This is so it can interact successfully with the IBM Update Utility.)

Fortran Preprocessor

```
        $JOB
        C OPTIONS N=90001,U=00000000,L=15,B='|',H='#',G='.',S=123,P=00,R=1,F= 3
        C###  UTEP FORTRAN PREPROCESSOR, VERSION 2.3,27 MAR 1978, EL PASO, TEXAS
        C###  DATE OF JOB: 02 MAR 1979, TIME: 17:15:47, STARTING LABEL: 90001
           (A number of source cards deleted)
        C      *** CHECK FOR JOKER
 113           IJ = 0
 114           IJOKE = 0
 115           FOR I FROM 1 TO 5 DO
 119              IF (IP(I) .EQ. 0)
 120              THEN
 122                 IJ = I
        C      .  .  *** MOVE JOKER TO 5TH POSITION
 123                 IR(IJ) = IR(5)
 124                 IS(IJ) = IS(5)
 125                 IR(5) = 0
 126                 IS(5) = 0
 127                 IJOKE = 1
        C      .  .  *** JUMP OUT OF CONTAINING LOOP
 128                 BREAK
 129              FI
 130           OD
 133           IF (IJ .EQ. 0)
 134           THEN
        C      .  *** NO JOKER
 136              ITYPE = JTYPE(IR, IS)
 137           ELSE
        C      .  *** HERE 5TH CARD IS JOKER
 139              ITYPE = 0
 140              FOR I FROM 13 + IACE TO 1 + IACE BY -1 DO
 144                 FOR J FROM 1 TO 4 DO
        C      .  .  .  *** TRY EACH POSSIBLE CARD
 148                    IR(5) = I
 149                    IS(5) = J
 150                    IT = JTYPE(IR, IS)
 151                    IF (IT .GT. ITYPE)
 152                    THEN
        C      .  .  .  .  *** HERE HAND BETTER THAN ANY PREVIOUS ONE
 154                       ITYPE = IT
 155                       IRJ = I
 156                       ISJ = J
 157                    FI
 158                 OD
 161              OD
        C      .  *** STORE BEST VALUE OF JOKER IN 5TH CARD
 164              IR(5) = IRJ
 165              IS(5) = ISJ
 166           FI
 167           RETURN
 168           END
```

FIGURE 2. WATFIV Compiler listing (recovered).

IV. DEVELOPMENT AND USE OF THE PREPROCESSOR

The preprocessor was coded in PL/I for use in a standard academic IBM environment (IBM 360/65 with OS/MVT and HASP). The first version was complete in just a few weeks as a PL/I program with some 400 statements. Then over a six-month period a great many changes and extensions were made to the original design, based on use of the preprocessor as it was changing.

The original version handled structured control structures similar to those in Figure 1. The changes and additions in chronological order were as follows:

1. An "INCLUDE" statement was incorporated. This had been planned from the beginning and was essential for large Fortran programs with many subroutines and numerous common blocks.

2. Some minor changes to the syntax made the control structures look more like those of Algol 68.

3. The "C OPTIONS" card was implemented as an improved way to feed options to the preprocessor.

4. The idea of a "recover" mode for the preprocessor was discovered and implemented.

5. The preprocessor was made to generate 8-digit update numbers in columns 73-80 of added text. These were needed for proper interfacing with the IBM Update Utility.

6. A reformatting feature was added to indent source statements in order to show the depth of nesting. The preprocessor uses a very simple reformatting algorithm: if there is insufficient room to move the statement over to the desired column, it just moves it as far as it can and makes no attempt to add another line. As an afterthought, optional insertion of grid marks was incorporated to show how many levels of nesting were present.

7. The generated standard Fortran statements in compiler listings were found to be of very little interest, so a special postprocessor for compiler listings was written to "recover" the source but retain line numbers and error messages of the compiler listing. This postprocessor will optionally list or suppress any statements incorporated with an "INCLUDE" statement.

Very few problems were encountered in implementing the original design. The "INCLUDE" feature was harder to implement than expected because our environment did not readily allow rewinding of certain source input files. The original design called for a final comment to be generated giving the total number of preprocessor errors (if there were any), as shown in Figure 3. This seemed straightforward and caused no problems with the WATVIV compiler. However, to the author's amazement, our version of the IBM Fortran H compiler gave a terminal error for this kind of final comment. Thus the preprocessor was changed so that it would not produce this comment for the H compiler.

The preprocessor has been used for three medium-sized Fortran development projects (each about 1000 card images long) and greatly facilitated the program development. The "INCLUDE" feature proved the most useful, since it allowed identical common blocks to be changed in a number of places at once. (In many environments, this feature will already be provided by some system utility.) The new control structures at hand allowed completely GOTOless programs to be produced, with statement labels only used for Format statements. There were essentially no problems with the preprocessor itself. All other forms of listing were quickly dropped in favor on the "recovered" compiler listing (illustrated in Figure 2), as soon as it was available.

The preprocessor was also used for part of two different university classes. One class was a liberal-arts mathematics class which learned some rudimentary programming using the preprocessor. Because list-directed input-output was available, the students were taught a version of Fortran with no statement labels at all! The other class was an engineering-oriented introductory Fortran class. The students were introduced to the preprocessor for one programming project. They adapted to the preprocessor more rapidly than was expected and even developed a very indignant attitude about the lack of an "IF-THEN-ELSE" statement in the standard Fortran they had been using. Preprocessor source errors which were properly identified by preprocessor error messages sometimes caused spurious compiler error messages which bothered some students.

```
      FOR I TO 100
C###  ### ERROR ("FOR" STMT MISSING "DO") ###
      FOR I TO 100 DO
      FI
C###  ### ERROR ("FI" WITHOUT MATCHING "IF") ###
      ELSE
C###  ### ERROR ("ELSE" ENCLOSED BY "DO-OD") ###
      OD
      OD
C###  ### ERROR ("OD" WITHOUT MATCHING "DO") ###
      ELSE
C###  ### ERROR ("ELSE" WITHOUT "IF-FI") ###
      FI
C###  ### ERROR ("FI" WITHOUT MATCHING "IF") ###
C###  ### 14 ERRORS IN THIS SUBPROGRAM ###
      END
C###  ### 15 PREPROCESSOR ERRORS TOTAL ###
```

FIGURE 3. Preprocessor error messages.

V. CONCLUSIONS

Let us first re-examine the list of disadvantages from section II and see whether this preprocessor takes care of them. The first three disadvantages relate to the need for double listings. With a "recovered" compiler listing (Figure 2), this preprocessor allows one to get along very well with just one short, easily read listing. The line number on the listing opposite an extended statement refers to the first of the sequence of generated statements which translate the source statement. In practice there was never any difficulty interpreting the line number references of a compiler error message. Thus the author feels that this preprocessor very successfully eliminated any need for double listings.

Furthermore, if the preprocessed text of a program is given to someone for use on another machine, it is readable in that form and a separate listing of the source text is not needed.

This preprocessor has extensive error checks and error messages (see Figure 3) but does not attempt any error correction. Actually, the reformatting was added as a partial error indicator since mistakes in the level of nesting were among the most common. With the reformatting, there was never any difficulty finding the cause of preprocessor errors.

Fortran Preprocessor

To help with disadvantage 5, the preprocessor copies a small part of the Algol 68 syntax. In a later version, one might replace "FI" with the "ENDIF" of Fortran 77. However, a problem with loops would remain, since the Fortran 77 "DO" still requires a statement label.

This preprocessor had execution time (disadvantage 6) generally somewhat less than compile time, but still significant. The preprocessor has very elaborate mechanisms designed to cut down on this extra preprocessor execution time. Unfortunately, these mechanisms proved to be a little too elaborate to be desirable in everyday use. Halfway through the development, when the "recover" mode was thought of, the idea was not to start always with the source as input to the preprocessor. A program development environment was visualized with frequent program updates. Suppose the preprocessed text is maintained as the main program text. Then preprocessing would not be needed if only a compilation was desired. Simple updates not involving extended features could also be made without running the preprocessor at all. Certain other simple updates might allow the preprocessor to be run in "normal" mode, where it would rapidly skip over any preprocessed text. For a complicated update the preprocessor could always be run in "recover" mode. Notice that for a run in "recover" mode, changes and additions could be made to the preprocessed text without regard to the location of any generated standard Fortran, since this would all be deleted anyway. In summary, the author feels that these elaborate features would eliminate much of the preprocessor execution time. However the simplicity of starting from the original source each time seemed to outweigh any savings obtained in this way.

Disadvantage 7, the fact that a preprocessor produces non-optimized code, would have to be lived with -- in the same way that we live with the inefficiencies of a high-level language compared with assembler language. (Of course we can always hand-code individual sections.)

PL/I was chosen because of its string-processing features and because it gave good access to the machine in our environment. Also this provides a degree of portability as PL/I becomes more widely implemented. In fact, an implementation in portable Fortran or in Snobol might prove rather unacceptably slow. (In our environment, PL/I is much faster for string processing than standard Fortran.)

In conclusion, Fortran can become almost pleasant when statement labels are needed only for Format statements. (Fortran 77 eliminates this need, but unfortunately does not have label-free loops.) In a large program environment, special routines to postprocess compiler listings seem particularly valuable and could be nicely combined with the type of preprocessor discussed here. Finally, automatic formatting seems especially desirable for program development, and with this preprocessor, using its "recover" feature, the source itself can be reformatted and not just the compiler listing.

REFERENCES

[1] D. E. Boddy, "Structured Fortran -- with or without a preprocessor," *SIGPLAN Notices*, vol. 12, no. 4, pp. 34-39, April 1977.

[2] R. Bond, "Free-form structured Fortran translator," *SIGPLAN Notices*, vol. 10, no. 10, pp. 12-15, Oct. 1975.

[3] S. Charmonman and J. L. Wagener, "On structured programming in Fortran," *SIGNUM Newsletter*, vol. 10, no. 1, pp. 21-23, Oct. 1975.

[4] P. Dirkson and I. McPhee, "Structured WATFIV and WATFOR-II," *Watnews*, Jan. 1977.

[5] L. E. Gales, "Structured Fortran with no preprocessor," *SIGPLAN Notices*, vol. 10, no. 10, Oct. 1975.

[6] D. S. Higgins, "A structured Fortran translator," *SIGPLAN Notices*, vol. 10, no.2, pp. 42-48, Feb. 1975.

[7] E. Horowitz, "Fortran -- can it be structured and should it be?" in *Practical Strategies for Developing Large Software Systems*, ed. by E. Horowitz, Reading, MA: Addison-Wesley, 1975.

[8] B. W. Kernigham, "RATFOR -- a preprocessor for a rational Fortran," Bell Laboratories, internal memorandum.

[9] C. L. McGowan and J. R. Kelley, *Top-down Structured Programming Techniques*, New York: Petrocelli/Carter, 1975.

[10] G. J. Myers, *Software Reliability: Principles and Practice*, New York: Wiley, 1976.

[11] A. Ralston and J. L. Wagener, "Structured Fortran -- an evolution of standard Fortran," *IEEE Trans. Software Engineering*, vol. SE-2, pp. 154-176, Sept. 1976.

[12] D. J. Reifer, "The structured Fortran dilemma," *SIGPLAN Notices*, vol. 11, no. 2, pp. 30-32, Feb. 1976.